Constraint Based Program Transformation Theory

Ph.D. Thesis

Stefan Natelberg

Software Technology Research Laboratory
De Montfort University, 10th November 2009

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Declaration

I declare that the work described in this thesis was originally carried out by me during the period of registration for the degree of Doctor of Philosophy at the De Montfort University, United Kingdom, from April 2006 to April 2009. It is submitted for the degree of Doctor of Philosophy at the De Montfort University. Apart from the degree that this thesis is currently applying for, no other academic degree or award was applied for by me based on this work.

Acknowledgements

First of all, I would like to acknowledge the financial support from Software Migration Limited and the De Montfort University for the research work.

My deepest gratitude goes to my supervisor, Professor Hussein Zedan. He was always very patient, gave me many inspiring advices and equipped me with the courage to go through one of the most challenging periods of my life yet make it one of my most important times as well.

I would like to thank my second supervisor, Dr. Martin Ward, for his precious advice and constant support. I am sincerely grateful to Professor Hongji Yang for his comments and suggestions about my work which were very helpful to me. Also, I appreciate Mike Dowd and Colin Bakker for their help with my work on various occasions.

Furthermore, I sincerely thank Professor Karl Hayo Siemsen for encouraging me to pursue my PhD degree. I am grateful to my colleagues at the University of Applied Sciences in Emden, Dr. Karsten Wolke, Dr. Konstyantin Yermashov and Andreas Rasenack.

A great many thanks go to my comrades at the De Montfort University, Matthias Ladkau, Peer Bartels, Dr. Feng Chen, Dr. Shaoyun Li, Sascha Westendorf, Keno Buss and many others. I thank them for their help and encouragement during the past years.

Finally, I am particularly grateful to my life companion Petra Schulte and my parents Christa and Heinz-Werner Natelberg. No words are adequate to express my appreciation for their love and support. They gave me the strength to go through ups and downs and helped me to be where I am now. This thesis is dedicated to them.

Abstract

The FermaT Transformation Engine is an industrial strength toolset for the migration of Assembler and Cobol based legacy systems to C. It uses an intermediate language and several dozen mathematical proven transformations to raise the abstraction level of a source code or to restructure and simplify it as needed. The actual program transformation process with the aid of this toolset is semi-automated which means that a maintainer has not only to apply one transformation after another but also to evaluate the transformation result. This can be a very difficult task especially if the given program is very large and if a lot of transformations have to be applied. Moreover, it cannot be assured that a transformation target will be achieved because it relies on the decisions taken by the respective maintainer which in turn are based on his personal knowledge. Even a small mistake can lead to a failure of the entire program transformation process which usually causes an extensive and time consuming backtrack. Furthermore, it is difficult to compare the results of different transformation sequences applied on the same program. To put it briefly, the manual approach is inflexible and often hard to use especially for maintainers with little knowledge about transformation theory.

There already exist different approaches to solve these well known problems and to simplify the accessibility of the FermaT Transformation Engine. One recently presented approach is based on a particular prediction technique whereas another is based on various search tactics. Both intend to automatise the program transformation process. However, the approaches solve some problems but not without introducing others. On the one hand, the prediction based approach is very fast but often not able to provide a transformation sequence which achieves the defined program transformation targets. The results depend a lot on the algorithms which analyse the given program and on the knowledge which is available to make the right decisions during the program transformation process. On the other hand, the search based approach usually finds suitable results in terms of the given

target but only in combination with small programs and short transformation sequences. It is simply not possible to perform an extensive search on a large-scale program in reasonable time.

To solve the described problems and to extend the operating range of the FermaT Transformation Engine, this thesis proposes a constraint based program transformation system. The approach is semi-automated and provides the possibility to outline an entire program transformation process on the basis of constraints and transformation schemes. In this context, a constraint is a condition which has to be satisfied at some point during the application of a transformation sequence whereas a transformation scheme defines the search space which consists of a set of transformation sequences. After the constraints and the scheme have been defined, the system uses a unique knowledge-based prediction technique followed by a particular search tactic to reduce the number of transformation sequences within the search space and to find a transformation sequence which is applicable and which satisfies the given constraints. Moreover, it is possible to describe those transformation schemes with the aid of a formal language.

The presented thesis will provide a definition and a classification of constraints for program transformations. It will discuss capabilities and effects of transformations and their value to define transformation sets. The modelling of program transformation processes with the aid of transformation schemes which in turn are based on finite automata will be presented and the inclusion of constraints into these schemes will be explained. A formal language to describe transformation schemes will be introduced and the automated construction of these schemes from the language will be shown. Furthermore, the thesis will discuss a unique prediction technique which uses the capabilities of transformations, an evaluation of the transformation sequences on the basis of transformation effects and a particular search tactic which is related to linear and tree search tactics.

The practical value of the presented approach will be proven with the aid of three medium-scale case studies. The first one will show how to raise the abstraction level whereas the second one will show how to decrease the complexity of a particular program. The third one will show how to increase the execution speed of a selected program. Moreover, the work will be summarised and evaluated on the basis of the research questions. Its limitations will be disclosed and some suggestion for future work will be made.

Publications

M.P. Ward, H. Zedan, M. Ladkau and S. Natelberg. Conditioned Semantic Slicing for Abstraction; Industrial Experiment. In *Software, Practice and Experience*, 38:1273-1304, October 2008

S. Natelberg. Constraint Based Program Transformation Theory. In *Proceedings of Informatiktage*. 6:77-80, March 2008

Table of Contents

List of Acronyms

AST	Abstract Syntax Tree
ANSI	American National Standards Institute
BNF	Backus-Naur-Form
CBPTS	Constraint Based Program Transformation System
CCM	Cyclomatic Complexity Metric
DFA	Deterministic Finite Automaton
DMS	Design Maintenance System
DMU	De Montfort University
FTE	FermaT Transformation Engine
GCC	GNU C Compiler
ICC	Intel C Compiler
ISO	International Organisation for Standardisation
JDK	Java Development Kit
LoC	Lines of Code
MCC	Microsoft C Compiler
NATO	North Atlantic Treaty Organisation
NoCC	Number of Code Characters

PC Personal Computer

PDL Pattern Description Language

SML Software Migration Limited

TSDL Transformation Scheme Description Language

UML Unified Modelling Language

WSL Wide Spectrum Language

ε-NFA Nondeterministic Finite Automaton which supports ε-Transitions

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Objectives

- To motivate the need for constraint based program transformation theory.
- To explain the research characteristics and to identify the research questions.
- To highlight the original contribution and to define the success criteria.

During the past few decades, the adaptation of software systems for a fast changing environment became more and more difficult. Nowadays, it is not only the technical and business-oriented requirements which drive these changes but also an increased pressure from the commercial side. In other words, programs have to be not only errorless and suitable for their corresponding task but also to be economical in acquisition, maintenance and sustenance. Therefore, the adaptation of software is crucial and must not be underestimated. The huge increase of program complexity which affects most of the available systems makes it even more difficult to adapt and extend such systems. For this reason, a lot of solutions have been presented for an automated program adaptation.

One of these solutions is the FermaT Transformation Engine (FTE) which has been developed by Martin Ward. This system is based on a set of mathematically proven transformations which are changing the source code of a program without changing its behaviour in terms of denotational semantics. Moreover, an intermediate language has been

developed which is particularly suitable for program transformations. This language is called Wide Spectrum Language (WSL) because of its wide abstraction spectrum. The theoretical foundation of the FTE and the basic principles of WSL will be discussed in Chapter 3 whereas an overview about the AST types which occur in this language will be provided in Appendix A. An overview about the FermaT transformations which will be used within this thesis is provided in Appendix A.

The FTE provides several possibilities to not only transform the WSL code but also to analyse it. Furthermore, some research has been carried out to automatise the process of software migration even more. This research is discussed in Chapter 2. With the original system, the maintainer has to apply one FermaT transformation after another. Moreover, he has to select the AST type on which the transformation will be applied as well. This can be a very difficult and frustrating task especially if the maintainer is not very familiar with transformation theory.

On the other hand, there is a prediction based approach as well as a search based approach. These approaches have been developed to solve the problem of manual transformation application and do neither require nor allow any interaction from the maintainer. Therefore, another problem arises because the maintainer is unable to take influence on the program transformation result which means he cannot bring his knowledge into the automated system. Furthermore, it is often difficult to find suitable transformation sequences without any restrictions because there are simply too many combinations of arranging the transformations within a sequence.

For this reason, the proposed approach of this thesis is constraint based and combines the advantages of manual, prediction based and search based program transformation. It uses an automata based program transformation process modelling technique in combination with constraints to achieve a restriction of the search space. Moreover, this approach hands the control over from the respective search tactic to the maintainer. The result is an adaptable, target-oriented prediction and search based program transformation approach which will be presented and discussed within this thesis.

1.1 Scope of the Thesis

In this thesis, a constraint based program transformation approach is proposed. This approach is based on the FTE as well as on WSL and combines automata theory, a prediction technique and a search tactic to provide a target-oriented program transformation system. It concentrates on the definition and classification of constraints as well as on constraint satisfaction via FermaT transformations and the modelling of program transformation processes. The scope of the thesis includes in particular:

1. The definition of constraints and constraint satisfaction and their relation to program characteristics and properties.
2. A classification of constraints into structural or behavioural constraints and low-level, high-level or environmental constraints.
3. The analysis of FermaT transformations and their application effects on the satisfaction of constraints.
4. The description of a FermaT transformation as a set of characteristics and properties for prediction purposes.
5. The modelling of the program transformation process via the introduction of transformation schemes.
6. The automated satisfaction of structural and behavioural constraints via a knowledge-based prediction technique followed by a search tactic.

1.2 Original Contributions

The original contributions of this thesis are as follows:

1. The first and most significant contribution is to motivate the need for a constraint based program transformation system and to define the three major parts which are the definition of constraints, the modelling of program transformation processes and the search for transformation sequences which application satisfies a set of given program constraints.

2. The second contribution is to define and classify program constraints on the basis of program characteristics and properties.

3. The third contribution is to restrict the huge search space of transformation sequences which often makes an extensive search on large systems infeasible. This restriction is achieved by program transformation process modelling as well as a prediction based approach which uses characteristics and properties of constraints.

4. The fourth contribution is to extend the operating range of the FTE to use it not only for software migration but also for other tasks of software optimisation and adaptation.

5. The fifth contribution is to provide a proper base for further research in the area of constraint based program transformation theory.

1.3 Research Questions

The following research questions are given to judge the success of the research which is described in this thesis:

1. Is it possible to model program transformation processes with the aid of maintainer knowledge?

2. How can the search space of existing search based approaches be restricted?

3. Is the proposed approach able to provide constraint-satisfying program transformation results in reasonable time?

4. How can constraints be integrated into program transformation theory?

5. What are the advantages of a constraint based program transformation approach against a common program transformation approach?

1.4 Organisation of the Thesis

The rest of the thesis is organised as follows:

- Chapter 2 gives an introduction to software evolution, provides an overview about program transformation approaches and discusses related projects, especially those which are based on target-oriented program transformation approaches.

- Chapter 3 gives an overview of the basic technologies on which the proposed approach of this thesis is based. This includes program transformation theory and the Wide Spectrum Language.

- Chapter 4 provides a definition and a classification of constraints and discusses the importance of structural and behavioural constraints for successful program transformation processes. It observes program transformation processes from different perspectives and discusses the relation of constraints to program characteristics and properties. Furthermore, the correlation of constraints with different pattern matching and measuring techniques are presented and explained.

- Chapter 5 discusses the capabilities and effects of FermaT transformations. It describes different influences of transformation applications on particular constraints and it explains how these effects can be captured and classified.

- Chapter 6 introduces transformation schemes and $\mathcal{META}$Constraints to model a program transformation process. Furthermore, it defines a formal language which is called Transformation Scheme Description Language (TSDL) to describe transformation schemes and it explains how these schemes can be constructed from the language.

- Chapter 7 discusses the application of transformation schemes which includes the decomposition of $\mathcal{META}$Constraints, the creation and reduction of the search space, the evaluation of the transformation sequences within the search space and the search for a particular applicable and constraint-satisfying sequence.

- Chapter 8 presents a system which has been developed to support this thesis. It gives an overview of the system architecture and discusses the main components.

- Chapter 9 provides three case studies on medium-scale WSL programs which prove the practical value of the proposed approach of this thesis.

- Chapter 10 provides a summary of this thesis, evaluates the research questions, discusses the limitations of the proposed approach and presents some suggestions

for the future work.

- Appendix A lists the AST types in WSL.
- Appendix B describes the FermaT transformations which have been used within this thesis.
- Appendix C presents the WSL code of the initial and final program of the second case study.
- Appendix D presents the WSL code of the initial and final program of the third case study.

1.5 Summary

The presented chapter has introduced constraint based transformation theory and explained the motivation and the targets of this thesis. Furthermore, it gave an overview about the original contributions, the research questions, the organisation and the scope of the thesis.

Chapter 2

Background and Related Research

Objectives

- To provide an introduction to software evolution.
- To discuss legacy systems and options to migration them.
- To give an overview of program transformation approaches.
- To identify difficulties and problems of the related research.

This chapter provides an introduction to the principles and problems of software evolution. It discusses legacy systems and options how these systems can be migrated. Furthermore, it gives an overview of existing program transformation approaches and discusses difficulties and problems of research projects which are related to the presented approach of this thesis.

2.1 Introduction to Software Evolution

Software evolution is the process of initial development of a software system followed by a maintenance phase to repeatedly change the software [41, 47]. According to Meir M. Lehman, this maintenance phase is necessary else the software becomes progressively less satisfactory [59]. On the other hand, it is often the case that changes increase the

complexity of the software. Moreover, it becomes less and less structured which makes further maintenance increasingly difficult. According to Frederick P. Brooks, there are a lot of real world examples where over 90% of the costs of a typical software system arise while in the maintenance phase [13]. The following sections provide an overview of software evolution and explain the basic principles.

2.1.1 Software Engineering

The term software engineering first came up in the late 1950s and early 1960s. Programmers have always known about civil, electrical and computer engineering and discussed what engineering might mean for software. In 1968 and 1969, the North Atlantic Treaty Organisation (NATO) Science Committee sponsored two conferences on software engineering which gave the field its initial boost. Many people believe that these conferences marked the official start of the profession [99].

Software engineering is a sub-discipline of computer science and deals with standardise manufacturing of software and the involved processes. The goal of software engineering is to develop high-quality software which is efficient, well structured and easy to maintain. There is no common definition of software engineering but one widely accepted definition was published by John N. Buxton, Peter Naur and Brian R. Randell in 1976 [15]:

> *"Software engineering is the establishment and use of sound engineering principles in order to economically develop software that is reliable and works efficiently on real machines."*

Software engineering is split into a large number of sub-disciplines which describe the development of software systems from scratch to the roll-out. These sub-disciplines can be summarised as follows [99]:

1. The elicitation, analysis, specification and validation of software requirements.
2. The creation of software designs on the basis of standardised formats such as the Unified Modelling Language (UML).
3. The construction of software with the aid of programming languages.

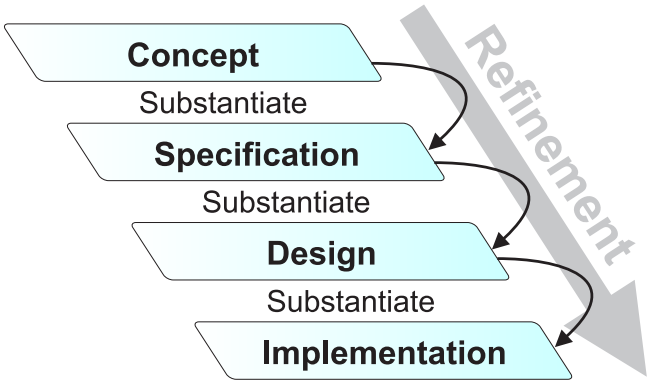
4. The testing of software as an empirical investigation to discover errors and to prove the quality of software.

5. The management of software configuration such as versioning and source control through standardised methods.

2.1.2 Software Development Process

The software development process tries to combine the sub-disciplines of software engineering into a layer-model where each of the process-layers contains one or more engineering tasks. Many different layer-models for all kinds of applications have been established but in general they can be divided into iterative and noniterative models. Noniterative models are passing through each layer just one time whereas iterative models pass the layers many times. A famous example for an iterative model is the Helix Model. A well known noniterative model is the Waterfall Model which is shown in Figure 2.1.

Figure 2.1: Software Development on the basis of a Waterfall Model [99].



2.1.3 Software Maintenance

Software systems were not very complex at the beginning of software development in the early 1950s and 1960s. The maintenance of software was just a small part of the software life cycle. With some revolutionary hardware developments in the late 1960s and early 1970s, software systems have become more complex and extensive [26]. Maintenance growth to a major part of the software life cycle. At that time, people first began to understand that software is not just dying. The re-engineering of software as sub-discipline of

software maintenance first came up.

At the end of the 1970s, the industry was faced with many major problems. The costs of software maintenance began to explode. In some areas, the costs of maintenance was taking more effort than the initial development. However, the problem of maintenance was caused by some properties and the complexity of software systems [55]. The demand for significant changes of software went up. The implementation of these changes themselves created additional problems [56, 58].

In general, software maintenance can be defined as the modification of a software product after delivery to correct faults, to improve performance or to adapt the product to a changed environment [99]. As mentioned before, the task of software evolution includes software development and maintenance so the term *software maintenance* is rarely used in literature.

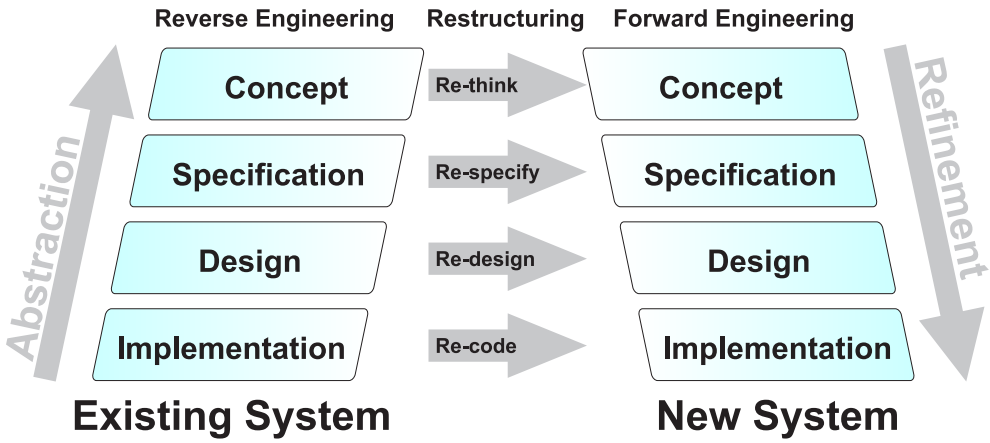
2.1.4 Software Re-engineering

Software re-engineering is a sub-discipline of software maintenance and is commonly known as the transposition and modernisation of software as well as its adaptation to new requirements. For example, this could be the porting of a software system to a new hardware platform, the correction of errors, the improvement of existing software features or the introduction of entirely new software functionalities [4]. In general, software re-engineering is a very difficult process depending on the complexity of the respective software system. It can be divided into the sub-disciplines reverse engineering, restructuring and forward engineering which will be discussed in the following sections. A well known general model of software re-engineering was published by Eric J. Byrne in 1992 [16]. This model is shown in Figure 2.2.

Reverse Engineering

According to popular apprehension, reverse engineering as sub-discipline of software re-engineering is a process to analyse and comprehend the components of software systems and their interrelationships. In general, this is achieved through the creation of system representations at a higher level of abstraction. For example, a representation can be a

Figure 2.2: General Model of Software Re-engineering [16].



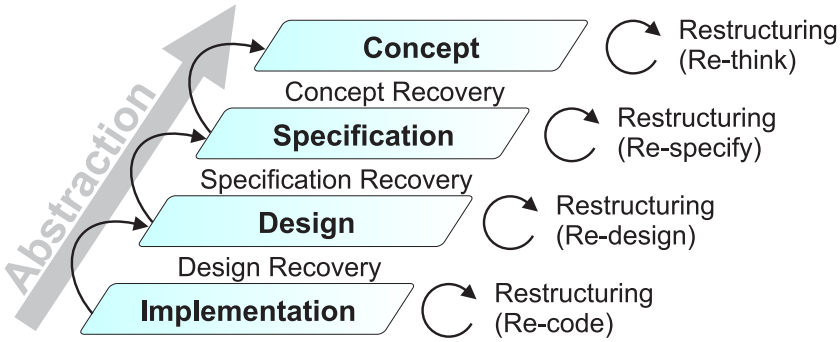
control flow graph, an UML diagram or even a written documentation of the software system.

In literature, reverse engineering has often been considered as the counterpart of forward engineering which is described in Section 2.1.4. For instance, Patrick Hall stated in 1992 that *reverse engineering can be seen as going backwards through the development cycle* [37]. However, a well-known definition of reverse engineering as applied to software was published by Elliot Chikofsky and James Cross in 1990 [19]:

> *"Reverse engineering is the process of analysing a [software] system to create representations of the system in another form or at a higher level of abstractions."*

Reverse engineering is often split into three subprocesses. These are the identification of system components, the analysis of the relation between these components and the analysis of the component functionalities. Moreover, there exist some layer-models which are related to the Waterfall Model. In general, the three described subprocesses have to be applied for each of the process-layers. Figure 2.3 shows a general model of reverse engineering.

Figure 2.3: General model of Reverse Engineering [99].



Restructuring

Restructuring as sub-discipline of software re-engineering is a process to transfer the source code of the initial program P_0 into the source code of the final program P_n. The abstraction level of the source code will be unchanged during this process. In general, restructuring has a broad and varied meaning and there is no common definition. However, one widely accepted definition was published by Jacques Eloff in 2002 [29]:

> *"Software restructuring is a form of perfective maintenance that modifies the structure of a ... source code."*

The aim of restructuring is often to increase the comprehensibility and the maintainability of a software system. This in turn simplifies the overall target of software re-engineering which is, as mentioned before, the transposition and modernisation of software as well as its adaptation to new requirements. Another aim of restructuring can be to change some of the program properties which includes the execution speed and the memory consumption of a software system. This will be discussed in Chapter 4.

In general, restructuring is carried out with the aid of automated program transformations which preserve the semantics of the respective program. A manual restructuring of software is difficult and may introduce undetectable changes in the behaviour of the program.

Forward Engineering

Forward Engineering as sub-discipline of software re-engineering is a process which is strongly related to the software development process. It also tries to combine the sub-disciplines of software engineering into a layer-model where each of the process-layers contains one or more engineering tasks. In general, forward engineering uses noniterative models which are passing through each layer just one time but this depends a lot on the particular case. The concept, the specification and the design of a software will be converted into a lower abstraction level during this process which is shown in Figure 2.1. The lowest abstraction level is the implementation and with it the source code of a program.

2.2 Characteristics of Legacy Systems

In terms of this thesis, a legacy system is an existing software system or a part of an existing software system which is relatively old but still in use. The documentation of a legacy system is often inconsistent or there is no kind of documentation at all. Furthermore, there is no original developer available. In fact, it is often the case that the source code of a program is everything that exists. Legacy systems usually share a lot of negative characteristics [99]. Some of the worst and typically ones can be summarised as follows:

1. Legacy systems are complex and large. The cyclomatic complexity of such systems can exceed the hundred and they often consist of more than a million Lines of Code (LoC). For example, a description of a legacy system migration has been published by Martin Ward in 1999 [87]. The legacy system which has been described in this paper has a cyclomatic complexity of 184 and consists of 5.9 million LoC.

2. Legacy systems are written in legacy languages such as Assembler or Cobol. For instance, the legacy system which has been migrated by Martin Ward in 1999 is written in Assembler [87] whereas another legacy system which has been migrated by Richard Millham in 2002 is written in Cobol [64].

3. Legacy systems are running on legacy hardware which has to be maintained at high costs. Moreover, legacy hardware is less effective in terms of power consumption compared to modern hardware systems. For example, a lot of legacy systems have been designed to run on a mainframe [66, 90].

4. Legacy systems are autonomous with little or no standardised interfaces to communicate with other applications. For instance, legacy systems do not provide a graphical user interface in most cases. Furthermore, a lot of such systems are based on a unique, proprietary database as well as network protocols which can be considered as legacy software as well [60].

5. Legacy systems are mission critical with availability rates at close to 100%. For example, such systems are often used to handle customer accounts in banks or insurance companies, to control nuclear power plants, to manage the air traffic or as computer reservation systems [72, 80].

The term legacy system has become well accepted within the software community. This is a result of the problems that legacy systems have brought to the computer industry and especially to the software sector. There exist a lot of legacy systems which have to be maintained, extended or redeveloped. These systems are usually very mature and the rare numbers of bugs are well known. The maintenance of such a system is difficult and expensive because of the complexity and the legacy language in which they are written. Moreover, it is not only expensive but also very dangerous to replace the complete system with a new one especially for business applications.

2.2.1 Legacy Systems written in Assembler Language

It is assumed that over 10% of the programs which are currently in operation are implemented in Assembler language [61]. These Assembler programs are often part of business critical and safety critical systems which are running on mainframes and have become legacy systems over time [71]. The percentage varies in different countries. A good example is Germany where it is estimated that about half of all data processing organisations are using Assembler systems [73].

It is very hard to maintain or extend such a system because an Assembler program is simply a list of instructions with labels and conditional or unconditional branches. The Assembler programmer has also to deal with registers and pointers which complicates the program code. The direct hardware access of the Assembler language can be executed very fast but causes some serious problems as well. For example, the identification of

all instructions which are branching to a particular label requires an analysis of the entire program.

On the other hand, a typical, well-designed program written in high-level language is much easier to comprehend. Conditional statements and loops are clearly indicated and variables are typed in almost all modern high-level languages. Such a language is more oriented on natural languages which makes it easier for a programmer or a maintainer to recognise standard pattern like an *IF* condition or a *WHILE* loop.

The Assembler language is not only harder to analyse than high-level language, there is also more code needed for a particular functionality. According to Capers Jones [44], a single function point requires about 575 lines of basic Assembler or 400 lines of macro Assembler to implement while only 220 lines of C or Cobol code are needed. A higher level language such as perl will require only 50 lines on average to implement one function point. All of these properties makes it difficult and expensive to maintain an Assembler program. Capers Jones Research [44] computed the annual cost per function point as follows:

1. Assembler: £48.00
2. PL/1: £39.00
3. C: £21.00
4. Cobol: £17.00

If a huge amount of Assembler code could be replaced by a smaller amount of well-designed and well-written high-level language code without seriously affecting the performance, the software maintenance costs could be much lower [85]. This and the decreasing pool of experienced Assembler programmers are increasing the pressure to move away from Assembler [97].

2.2.2 Dealing with Legacy Assembler Systems

There are a number of options for dealing with legacy Assembler systems. To a high degree, it depends on the circumstances which of these options is the best for a legacy Assembler system. Each option can also provide problems and risks which will be discussed in the following sections [97].

Keep the Current System

It is always a possibility to keep the current application if it is relatively stable. To build a new system around the existing Assembler software and leave the rest untouched is relatively cheap and will probably not cause much errors. If there are experienced Assembler programmers available, it should also be possible to extend the existing software system even if it has already become a legacy system. In this case, an automated Assembler analyses could help to comprehend the code. On the other hand, the hardware on which a program is supposed to run is often more problematic. It might be necessary to extend the hardware after the software system has been modified.

Replace the Current System with a Vendor Package

According to Brooks [13], the most radical possible solution for constructing software is not to construct it at all. The option to replace an existing legacy system with a vendor package would probably solve the maintenance issue. The actual problem is to find a vendor package which exactly meets the requirements and does not have many differences compared to the existing software system [54]. Furthermore, it is often the case that an existing business model has to be adapted to a new application. This can be significant if the existing model provides advantages compared to business models of the competitors.

Rewrite the Current System in High-Level Language

In many cases, a system has been extended a lot if it is in use over many years. Usually, these extensions do not exactly fit into the original system design. The structure of the Assembler code has become very complex and the costs to maintain or extend it are too high. An additional problem is that there are often no experienced Assembler programmers available. In this case, it can be an option to rewrite the Assembler system in a high-level language. Nevertheless, the costs to comprehend the existing system, to develop a new system design and to implement the new system in a modern programming language can be enormous. Furthermore, the effort to test and debug this new system must not be underestimated.

Migrate the Current System to High-Level Language

Most of the legacy systems which are currently in use are working satisfactorily for their customers. There are often just a few adaptations or extensions to make. It is also possible

that the customer wants to migrate to another platform because the original hardware is too expensive or no longer available. In this cases, it would be very uneconomical to rewrite the entire legacy software. An automated migration technology to transfer the Assembler software to high-level language could be a better solution.

2.3 Overview of Program Transformation Approaches

Nowadays, there exist a lot automated migration systems [65]. Some of them are working with brute force algorithms which simply map each Assembler instruction to an equivalent statement in the high-level language [20]. This kind of migration has many disadvantages which can be summarised as follows:

1. The resulting high-level code is very complex and unstructured which makes program comprehension and maintenance more difficult.
2. The resulting program is about seven times slower than the original program [32].
3. The possibilities of the target language to improve maintainability and high-integrity are often underutilised.

The complex and unstructured high-level code which is generated by brute force translators is often useless for systems which are frequently maintained. It is also useless for program design comprehension. Actually, it is only useful for small applications which do not depend on execution speed and which have to be migrated to a different platform for a particular reason [97].

2.3.1 FermaT Transformation Engine

The FermaT Transformation Engine (FTE) is based on a different approach. The Assembler code of a legacy system will be translated into an intermediate language called WSL [91]. The translation process is based on a brute force algorithm where each Assembler instruction is mapped to an equivalent statement in WSL. This intermediate language is uniquely suitable to apply FermaT transformations [88]. Once the Assembler code has been translated, various FermaT transformations will be applied to restructure and to simplify the resulting WSL code and to raise its abstraction level [97]. Afterwards, the transformed WSL code can be translated into the target language. This approach does not

have the huge disadvantages of the brute force approach but other problems arise.

A program transformation process with the aid of the FTE is semi-automated which means that a maintainer has not only to apply one FermaT transformation after another at a particular AST path. He has also to evaluate the transformation result. This can be a very difficult task especially if the given program is very large and if a lot of FermaT transformations have to be applied [92]. Moreover, it cannot be assured that a transformation target will be achieved because it relies on the decisions taken by the respective maintainer which in turn are based on his personal knowledge. Even a small mistake can lead to a failure of the entire program transformation process which usually causes an extensive and time consuming backtrack. Furthermore, it is difficult to compare the results of different transformation sequences which are applied on the same program. These problems could be solved by the use of a reliable prediction technique, an effective search tactic or a combination of both. This has to generate an applicable transformation sequence which leads to the target of the program transformation process. The constraint based program transformation approach which is proposed in this thesis is based on a combination of both, a prediction technique as well as a search tactic.

2.3.2 Design Maintenance System

The FTE is just one of many existing program transformation approaches. Another one is the Design Maintenance System (DMS) [9]. This approach is currently being implemented with the target to manage software evolution on the basis of automated tools. The Transformation Engine is one of these tools. It contains a collection of individual transformations which are called transforms in this context. These transforms are comparable to FermaT transformations. Ira D. Baxter, Christopher Pidgeon and Michael Mehlich described transforms in 2004 as follows [11]:

> *"... transforms are generally described as source-to-source rewrite rules. In theory, however, transforms are simply functions from program representations to program representations."*

The DMS approach has not been developed to abstract or refine a program. The purpose of this approach is to restructure the source code of a program as needed or to perform predictable semantical changes. To achieve this, it is not using a particular transformation language as is the case with the FTE. Instead, it provides various techniques

to define and implement parsers. Therefore, a particular parser which is able to generate an AST for a particular program has to be implemented for each programming language [11]:

> *"An extremely practical technology is ... [the] ability [of DMS] to explicitly define languages, parse programs [which are written] in those languages and build abstract syntax trees for these programs ..."*

Despite the fact that transforms are not based on a particular transformation language, the implementation of these transforms is comparable to the implementation of FermaT transformations. Both approaches consider a transformation as an operation on an AST. Furthermore, both approaches use conditions to determine if a transformation is applicable at a particular place within the program. Therefore, transforms are related to FermaT transformations in terms of their basic functionality and their application [10]:

> *"... [the program] transformation system will have a large number of [transforms] ... and [there are] a large number of possible places ... [within] a program to apply them."*

The large number of transforms and the great variety of possible places to apply them within a program lead to problems which are similar as is the case with the FTE. As mentioned before, a reliable prediction technique, an effective search tactic or a combination of both could be used to solve these problems.

2.3.3 Stratego/XT

Stratego/XT is another program transformation approach. The basic implementation of this approach has already been finished but it is continuously maintained and enhanced. The Stratego/XT implementation provides a collection of transformation tools referred to as XT. It also contains a set of individual transformations which are called transformation rules in this context. These transformation rules are comparable to FermaT transformations. Eelco Visser described transformation rules in 2004 as follows [81]:

> *"A transformation rule encodes a basic transformation step as a rewrite on an abstract syntax tree."*

Similar to the DMS approach, the Stratego/XT approach has not been developed to abstract or refine a program. More precisely, there is no intended purpose of this approach at all. It serves more or less as framework to implement and perform program transformations for various objectives [12]. However, transformation rules can only be applied on programs which are written in a particular language. This language is called Stratego and was explicitly developed to support program transformations. The approach of an explicit transformations language is similar to the WSL approach but it lacks the support of the wide abstraction spectrum.

The implementation of transformation rules in turn is a lot more flexible than the implementation of FermaT transformations or transforms. A transformation rule can be defined as an operation on an AST but it can also be described by using the concrete syntax of the Stratego language. Despite this relatively flexible implementation, transformation rules use conditions to determine if a transformation is applicable at a particular place within the program. Therefore, transformation rules are related to FermaT transformations and transforms in terms of their basic functionality and their application [82]:

> *"... instead of applying transformation rules throughout the subject program, we often wish to apply them locally ... to select parts of the subject program. This allows us to use transformation rules that would not be beneficial if applied everywhere."*

In this context, the subject program is the program which has to be transformed. Accordingly, the Stratego/XT approach supports global and local transformation rules. The great variety of possible places to apply local transformation rules within a program and the large number of global and local transformation rules lead to problems which are similar as is the case with the FTE and with the DMS. As mentioned before, a reliable prediction technique, an effective search tactic or a combination of both could be used to solve these problems.

2.4 Search based Program Transformation

The proposed approach of this thesis is based on the FermaT Transformation Engine (FTE) which is an industrial strength toolset for the migration of Assembler and Cobol based legacy systems to C. As mentioned before, a program transformation process and

therefore the migration of a legacy system with the aid of this toolset is semi-automated which causes a lot of problems. There exist two major approaches to solve these problems. The first one uses hill-climbing and genetic search tactics to generate an applicable transformation sequence which leads to the target of the program transformation process. This approach has been published by Deji Fatiregun, Mark Harman and Robert M. Hierons in 2003 [30].

2.4.1 Program Transformation as a Search Problem

In general, a program transformation process uses various FermaT transformations to convert the initial program P_0 into the final program P_n. Accordingly, the initial program P_0 is the first program state of a program transformation process whereas the final program P_n is the last program state of such a process. If the transformation sequence which will be applied during this process consists of more than one FermaT transformation, there also exist a number of program states $P_1 \ldots P_{n-1}$.

Since a FermaT transformation can be considered as an operation on an AST, each of the program states $P_0 \ldots P_{n-1}$ has to be parsed and an AST has to be generated. An AST of a program state P_i in turn contains AST types as nodes and leafs. Each of these types is either a general type, a group type or a specific type and is nested according to the rules of the WSL language. These rules will be described in Appendix A.

A particular FermaT transformation can be applied on a number of selected AST types within a program state P_i. This is determined by the applicability condition of the respective transformation. Therefore, there exist a set of possible triplets during a program transformation process which can be considered as the search space. Each triplet $\Gamma(\alpha, \beta, \chi)$ within the set consist of:

1. (α) an applicable FermaT transformation,
2. (β) a program state P_i which has to be parsed and
3. (χ) a place within the program state P_i where the transformation will be applied.

This is just one of many possibilities to define the search space of a program transformation process. Another possibility is to describe it as a set of transformation sequences

Ω where each FermaT transformation has to be applied on various places within a program state P_i. The order in which these transformations are applied is of course important to achieve the desired program transformation target.

No matter how the search space is described, it is often very large. This is the greatest disadvantage of search based approaches and makes exhaustive search infeasible in a lot of cases [31]:

> *"Given a ... system with 20 possible [FermaT] transformations and a [fixed] sequence length of 20, there are* 20^{20} *possible combinations in the search space ..."*

However, this assumption is still very optimistic. It does not take the places into account where the FermaT transformations can be applied within a program. To consider these places would lead to an even larger search space. Nevertheless, there are two search based approaches which have been proven to be relatively efficient [31].

2.4.2 Hill-Climbing Search Tactic

A hill-climbing search tactic is a heuristic optimisation method. Before this search tactic can be performed, a transformation sequence within the given search space has to be randomly selected. This sequence can be considered as the starting point of the search. Furthermore, the algorithm which represents the implementation of the search tactic has to be able to evaluate a transformation sequence on the basis of a fitness function. In the approach of Deji Fatiregun, Mark Harman and Robert M. Hierons, this function simply counts the Lines of Code (LoC) of the final program P_n where less lines means a higher fitness. Afterwards, the algorithm checks each neighbour of the selected sequence and changes the position to the fittest sequence if it is better than the selected one [68]. For example, the neighbours of a sequence with the index i within a linear search space are the sequences with the index $i-1$ and with the index $i+1$. The position will be changed as long as any better neighbour can be found. If there is no better neighbour, the algorithm assumes that it has arrived at the top of a hill and the current solution remains the best. The search tactic can be performed several times with the aim to divert the algorithm from any local optima and to increase the chances to find a global solution.

Listing 2.1 shows the algorithm of a simple tactic that works on the basis of the one-dimensional search space Ω. Nevertheless, hill-climbing search tactics can be used in combination with multi-dimensional search spaces as well. The presented algorithm uses two methods to keep it as simple as possible:

1. **int nextInt(int n)** - Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value *n* (exclusive).

2. **int getFitness(int i)** - Returns the fitness of the element with the specified index *i*.

The algorithm consists of only one method and is written in Java. The integer variable *i* represents the index of the selected sequence whereas the integer variable *f* is the fitness of the sequence with the index *i*. Since the algorithm works on the basis of a two-dimensional search space, it chooses to iterate either the right or the left side of the transformation sequence.

Listing 2.1: Hill-Climbing Algorithm in Java.

```
1  public int getLocalMaximum(Sequence sequence) {
2
3    /* local variables */
4    int i = new Random().nextInt(sequence.length() - 1),
5        f = sequence.getFitness(i);
6
7    /* iterate the right side */
8    if (i == 0 || sequence.getFitness(i - 1) <
9      sequence.getFitness(i + 1))
10     while (sequence.getFitness(i + 1) > f)
11       f = sequence.getFitness(i++);
12   /* iterate the left side */
13   else
14     while (sequence.getFitness(i - 1) > f)
15       f = sequence.getFitness(i--);
16
17   return i;
18 }
```

2.4.3 Genetic Search Tactic

A genetic algorithm is a population-based search tactic which starts with an initial random population and evolves over several generations. The individuals in successive generations in turn have better or at least no worse fitness values than those in preceding generations [21]. An optimised individual is one which presents a more desirable solution to the given problem.

Genetic algorithms imitate the natural process of evolution with the aid of the operators *crossover*, *selection* and *mutation*. These so-called evolutionary operators are used to alter the population across several generations toward optimality [14]:

1. *Selection* is the process of choosing two individuals on the basis of their fitness. These individuals are needed for crossover.

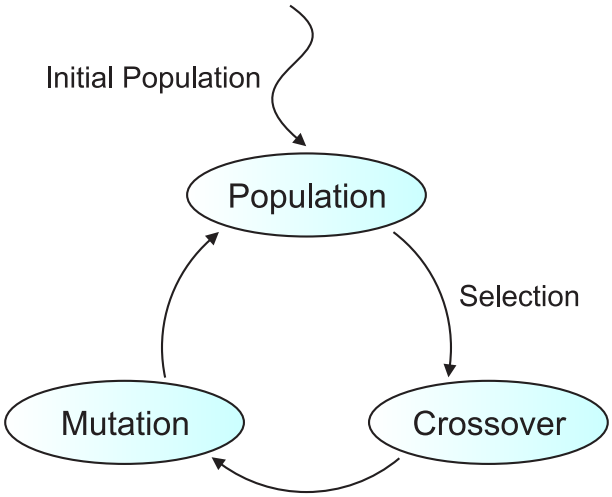
2. *Crossover* is the process of exchanging information between two individuals. This will be achieved by dividing each individual at a selected position and swapping adjacent sides across to create new child individuals.

3. *Mutation* is the process of introducing diversity into the population. This will be achieved by providing a chance for a chromosome to be altered. The probability that a chromosome will be altered is referred to as the mutation rate [22].

The evolutionary operators are applied iteratively across each new population where the genetic algorithm will terminate after a finite number of generations. Figure 2.4 shows the iterative stages of a genetic algorithm.

The approach of Deji Fatiregun, Mark Harman and Robert M. Hierons uses a genetic algorithm to evolve transformation sequences. These sequences will be applied on the initial program P_0 and are supposed to lead to the target of a program transformation process. To achieve this, a transformation sequence has been defined as a particular individual which has to be optimised. Accordingly, a single FermaT transformation can be considered as chromosome whereas the entire population can be regarded as set of transformation sequences. Each time a parent sequence is applied to the initial program P_0 its fitness is evaluated. The same applies to every child sequence which is generated during the crossover and re-admitted into the population. To use a transformation sequence as an individual simplifies the definition and the implementation of the evolutionary operators:

Figure 2.4: Iterative stages of a Genetic Search Tactic [31].



1. *Selection* has been implemented with the aid of the fitness function which evaluates the transformation sequences. This fitness function in turn is based on an algorithm which simply counts the LoC of the final program P_n where less lines means a higher fitness. Only the fittest transformation sequences will be selected for crossover.

2. *Crossover* has been implemented as the combination of two transformation sequences at a single, randomly selected position. The sequences will be divided at this position and the adjacent sides will be swapped. The result are two transformation sequence which are possibly not applicable.

3. *Mutation* has been implemented as an exchange of a randomly selected FermaT transformation within a particular transformation sequence. The result is a transformation sequence which is possibly not applicable.

However, the genetic search tactic provides far better results than the hill-climbing algorithm [31]. Nevertheless, even a genetic search tactic is not able to deal with the huge search space of common program transformation processes.

2.5 Prediction based Program Transformation

As mentioned before, there exist two major approaches to solve the problems which arise during a program transformation process with the aid of the FTE. The first one uses hill-climbing and genetic search tactics to generate an applicable transformation sequence which leads to the target of the program transformation process. However, the huge search space makes this approach inappropriate for large program transformation processes. For this reason, Shaoyun Li proposed a prediction based approach which uses software metrics to model program transformation targets and a complex prediction technique to calculate applicable transformation sequences which lead to this target.

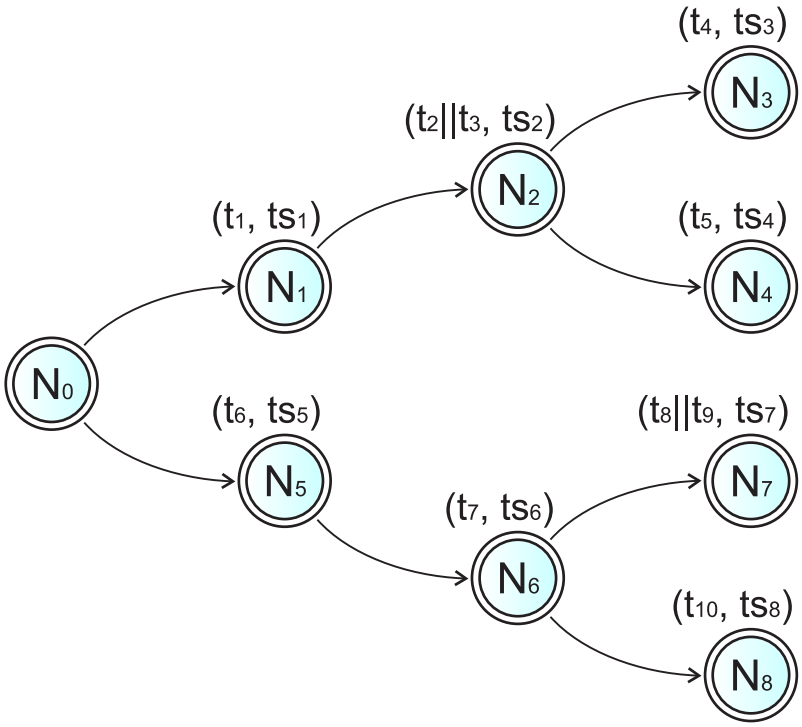
2.5.1 Transformation Process Models

The approach is based on so-called transformation process models to calculate applicable transformation sequences. Afterwards, these sequences have to be evaluated in terms of the defined target. The model is represented as a directed tree with transformation steps as nodes. At each of these steps, at least one transformation will be applied on the given program. Figure 2.5 shows an example of a transformation process model where a transformation step is represented by a double-circle.

The model defines various transformation sequences which are represented as different paths within the tree. These paths are traversed from the root node to a final node of the model where a node is considered as final if it has no outgoing edges. In other words, the prediction technique is working with a set of current program versions V_i on which different FermaT transformations will be applied. Afterwards, it evaluates the transformation results with the aid of different measuring techniques. For this reason, an evaluation system has been developed which rates the effect of each applied FermaT transformation in terms of the respective targets. If the effect of a transformation is not positive, the resulting program version will be discarded. This is supposed to lead to a linear improvement of the program versions in relation to the defined targets.

In general, the approach of Shaoyun Li can be executed a lot faster than extensive search based approaches. Nevertheless, the prediction based approach has some disadvantages as well. It is not guaranteed that a program transformation process which is based on this approach will lead to an improved program version in terms of the defined

Figure 2.5: Example of a Transformation Process Model.



targets. In fact, it cannot be guaranteed even if there exist a transformation sequence which leads to this target and which can be generated with the aid of the given FermaT transformations. This is because the approach ignores some of the given transformations to reduce the size of transformation process models [60]:

> *"... [FermaT] transformations which will not change the source code features [in terms of the defined targets] will not be taken into account."*

To ignore FermaT transformations which do not provide a direct benefit leads to the fact that the approach will not deliberately exploit interplay effects. For example, an interplay effect causes that an application of a particular FermaT transformations prepares the program for a subsequent application of another transformation. But if the application of the first transformation does not change the source code features, it will be ignored by the respective transformation process model. Accordingly, the following transformation which might be important to achieve the overall transformation result cannot be applied. The following listing shows an example WSL program.

Listing 2.2: WSL Example Program to Demonstrate Interplay Effects.

```
1 DO
2   SKIP;
3   DO
4     IF i = j THEN
5       EXIT(2)
6     FI;
7     i := j
8   OD
9 OD
```

The presented program contains a *DO* loop which in turn contains a *SKIP* statement and another *DO* loop. To remove one of the loops is an easy task and simplifies the program a lot. However, the *SKIP* statement has to be removed first to apply the following FermaT transformation. Therefore, the Delete All Skips transformation has to be applied on the root type of the program followed by the Double to Single Loop transformation which has to be applied on the first specific type T_Floop. The following listing shows the resulting WSL program.

Listing 2.3: Transformed WSL Example Program to Demonstrate Interplay Effects.

```
1 DO
2   IF i = j THEN
3     EXIT(1)
4   FI;
5   i := j
6 OD
```

As mentioned before, the prediction based approach is not able to recognise the interplay effects between two FermaT transformations. This means that it would have problems to improve the presented WSL program if, for example, the transformation target would be to decrease the nesting depth of the program.

2.5.2 Transformation Composition

As mentioned before, a transformation process model contains transformation steps. Each of these steps can be considered as a particular FermaT transformation or a set of FermaT

transformations. If it is a set, the transformations are composed on the basis of a specific operator [60]:

> *"Parallel composition (||) will be applied where a set of [FermaT] transformations can be executed in parallel. For example, $t_1||t_2$ expresses that [transformation] t_1 and [transformation] t_2 can be applied in parallel on independent program blocks."*

In this context, parallel execution is not considered in terms of time. It proposes that a couple of FermaT transformations with limited reach of efficacy can be executed as one step without analysing the AST of the given program again. In fact, this leads to a lot of problems. One major problem is to define the reach of efficacy for each FermaT transformation. For instance, there are a lot of transformations like the Absorb Left or the Constant Propagation which can have a massive influence on the entire program. Therefore, it cannot be guaranteed that the approach is suitable for each of the FermaT transformations. Another major problem is that even FermaT transformations with a very small and predictable reach of efficacy can change the AST path on which another transformation is supposed to be applied. The following listing shows an example WSL program on which the transformation composition ($t_1||t_2$) will be applied to reveal the problem. For a better comprehension, the AST paths of the individual assignments are stated in curly brackets.

Listing 2.4: WSL Example Program to Demonstrate the Weaknesses of Transformation Composition.

```
1 i := 1;          {/0/}
2 j := i + i + i   {/1/}
```

If the transformation t_1 is the Add Assertion applied on the AST path /0/ and the transformation t_2 is the Simplify Item applied on the AST path /1/ of the presented WSL program, there will occur an error. The first FermaT transformation will introduce an assertion at AST path /1/ as shown in the following listing. For a better comprehension, the AST paths of the individual assignments and of the introduced assertion are stated in curly brackets.

Listing 2.5: Transformed WSL Example Program to Demonstrate the Weaknesses of Transformation Composition.

```
1 i := 1;          {/0/}
```

```
2 {i = 1};          {/1/}
3 j := i + i + i    {/2/}
```

After the first FermaT transformation has modified the program, the second FermaT transformation will be applied on the assertion and not on the second assignment as it was supposed to be.

2.6 Summary

The presented chapter has discussed the related work of this thesis. It has provided an introduction to software evolution and described the characteristics of legacy systems. Furthermore, it gave an overview of program transformation approaches and discussed a search based as well as a prediction based approach which address some problems of program transformation processes.

Chapter 3

Preliminaries

Objectives

- To introduce program transformation theory.
- To discuss the basic principles of the Wide Spectrum Language.
- To describe the implementation of FermaT transformations.

This chapter discusses the theoretical foundation of the FermaT Transformation Engine (FTE). It provides an introduction to program transformation theory and explains the basic principles of the Wide Spectrum Language (WSL). Furthermore, it describes the implementation of FermaT transformations on the basis of a transformation description language which is called $\mathcal{M}_{ETA}$WSL.

3.1 Introduction to Program Transformation Theory

The term program transformation describes the generation of a target program P' from a source program P [89]. In general, it is required that both programs are semantically equivalent in terms of a particular formal semantics but there are cases where a predictable semantical difference is desired [91]. Program transformation is useful for various applications in the area of software evolution. This includes program comprehension, reverse

engineering and compiler optimisation [21, 22].

There are a few different methods to perform extensive program transformation processes. All of the program transformation approaches which have been discussed in Chapter 2 are based on a particular one. These approaches split an entire program transformation process into a sequence of basic transformations which have to be applied at a particular place within the program [91, 10, 82].

The fact that three approaches are based on the same method to perform program transformation processes leads to similarities of their implementations. This applies especially in regard to the FTE and Stratego/XT. Both approaches provide a transformation language as well as some tools to parse this language and to generate a corresponding AST. The generation of an AST is necessary because both approaches define at least some of the available transformations as an operation on such an AST. There are also some tools to translate the transformation language to other languages and vice versa. The DMS in turn does not provide such a language but it offers various techniques to define and implement parsers. Each of these parsers is able to generate a corresponding AST for programs which are written in a particular programming language. A property which is shared by the implementation of all of the three approaches is an extensive collection of transformations, a tool to apply these transformations and a framework to implement them. Since the proposed approach of this thesis is based on the FermaT Transformation Engine (FTE), the theoretical foundation of this approach will be discussed in the following sections.

3.1.1 Definition of Semantic Equivalence

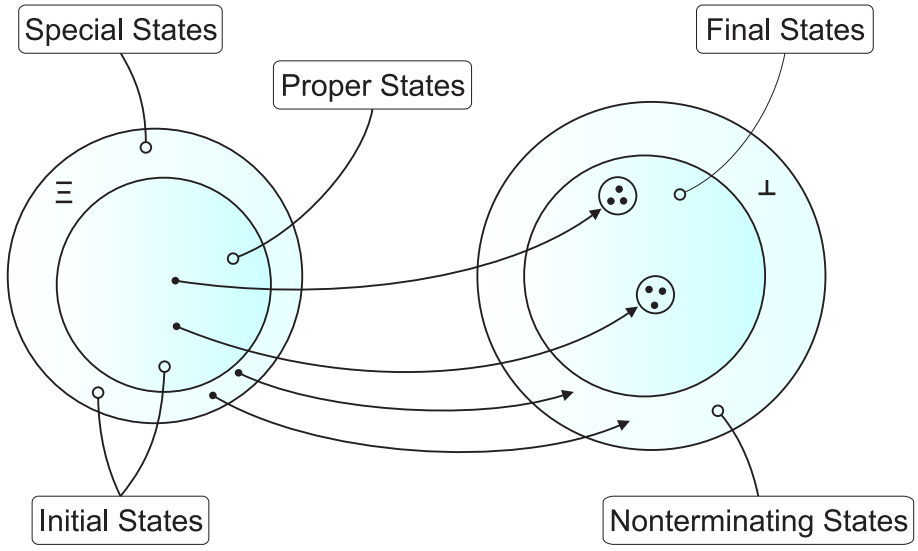
Each of the FermaT transformations has been mathematically proven to generate a target program P' which is semantically equivalent to the source program P. To provide this proof, a particular mathematical model has been developed. Furthermore, the meaning of the term semantic equivalence has been precisely defined with the aid of this model. A description of it was published by Martin Ward in 2007 [96]:

> *"... [the] mathematical model [which is used to prove FermaT transformations] defines the semantics of a program as a function from states to sets of states. For each initial state s, the [semantic] function f returns the set of*

states $f(s)$ which contains all the possible states of the program when it is started in state s."

The mathematical model is based on the theoretical foundation of denotational semantics which has been published by Robert E. Tennent in 1976 [78] and by Joseph E. Stoy in 1981 [76]. This semantics ignores the internal sequence of state changes that a program carries out [100]. Figure 3.1 shows the semantics of a WSL program.

Figure 3.1: Semantics of a WSL program [50].



Each initial state *s* which possibly leads to nontermination is indicated as a special state Ξ where nontermination itself is indicated as $\perp$. All other initial states are often referred to as proper states. The mathematical model which is used to prove FermaT transformations always considers two special states as semantically equivalent [95]:

"If $\perp$ is in the set of final states [$f(s)$] then the program might not terminate for that initial state [s]. If two programs are both potentially nonterminating then we consider them to be [semantically] equivalent on that state."

The restriction of the mathematical model simplifies the process of proving the semantic equivalence of two programs which in turn simplifies the development of FermaT transformations. On the other hand, the restriction leads to the fact that the mathematical

model does not distinguish between an initial state which might not terminate and an initial state which never terminates. In fact, Martin Ward once claimed that *a program which might not terminate is no more useful than a program which never terminates* [96]. However, the semantic equivalence of programs has been defined on the basis of the semantic function f [89]:

> *"If two programs have the same semantic function [f] then they are said to be [semantically] equivalent."*

To put it briefly, two programs are said to be semantically equivalent if the semantic function f of program one and program two return the same set of states $f(s)$ for each initial state s.

3.1.2 Program Abstraction and Refinement

The abstraction and the refinement of a program is not only one of the main tasks of the FTE. It is also the reason why WSL has been developed. This particular transformation language provides a wide abstraction spectrum and therefore allows to apply transformations which change the abstraction level of a program.

According to popular apprehension, the term program abstraction describes the generalisation of a program. A widely accepted definition of this process was published by Martin Ward in 1995 [86]:

> *"Program abstraction is a process of generalisation, removing restrictions, eliminating detail and deleting inessential information such as the algorithmic details."*

The FTE provides various FermaT transformations to carry out a program abstraction. One of them is the Program to Specification transformation which converts a common WSL program into a WSL specification. This is helpful for reverse engineering as well as for program analysis and comprehension because a specification only defines *what* a program does without describing *how* it is done. The following listing shows a simple WSL example program on which the Program to Specification transformation can be applied.

Listing 3.1: High-Level WSL Program Example.

```
1 IF i > 25 THEN
2   j := 0
3 ELSIF i > 10 THEN
4   j := 1
5 ELSIF i < 10 THEN
6   j := 2
7 ELSE
8   j := 3
9 FI
```

The program consists of an *IF* statement which contains the variable *i* and the variable *j*. This representation is on the same abstraction level as programs which are written in languages like C or Java. Therefore, it can be considered as high-level language. After the application of the Program to Specification transformation, the resulting program is semantically equivalent but has been converted to a higher abstraction level. The following listing shows this program.

Listing 3.2: WSL Specification Example.

```
1 SPEC <j>:
2   j' = 1 AND i > 10 AND i <= 25 OR
3   j' = 0 AND i > 25 OR
4   j' = 2 AND i < 10 OR
5   j' = 3 AND i = 10
6 ENDSPEC
```

The FTE provides not only various FermaT transformations to carry out a program abstraction. It provides also some FermaT transformations to refine a specification or a program which can be considered as the opposite of abstraction. For example, the Refine Specification transformation is the converse of the Program to Specification transformation. Accordingly, it converts a WSL specification into a WSL program.

In general, the process of program refinement is a lot more popular than the process of program abstraction [86]. It describes the substantiation from a specification through to the executable of a program [5, 6]. For instance, it is necessary to carry out program refinement for each software development process which has been discussed in Chapter

2. A well known definition was published by Ewen W.K.C. Denney in 1998 [25]:

> *"Program refinement is a programming methodology in which a formal description of what a program should do - a specification - is gradually refined into an executable program satisfying that specification."*

As mentioned before, two programs are said to be semantically equivalent if the semantic function f of program one and program two return the same set of states $f(s)$ for each initial state s. In terms of the mathematical model which is used to prove FermaT transformations, refinement can be considered as a generalisation of semantic equivalence [96]. A refined program is more defined and more deterministic than the original specification or program. Therefore, Martin Ward described a refinement in terms of the FTE as follows [95]:

> *"One program is a refinement of another if it terminates on all the initial states for which the original program terminates ... [and] for each such state it is guaranteed to terminate in a possible final state of the original program."*

If the program P has the semantic function f, the program P' has the semantic function f' and $f'(s) \subseteq f(s)$ applies for all initial states s then the program P is refined by the program P' or the program P' is a refinement of the program P.

3.1.3 Validity of FermaT Transformations

The validity of FermaT transformations can be proven with the aid of the predicate transformer semantics approach which has been published by Edsger W. Dijkstra in 1975 [27]. This approach has been developed to verify the correctness of computer programs [69]. Predicate transformer semantics is an extension of the Floyd-Hoare Logic which in turn is a formal system that contains a set of logical rules for reasoning about computer programs [28]. The Floyd-Hoare Logic was developed by Charles A.R. Hoare in 1969 with the intent to *explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics* [39]. Edsger W. Dijkstra described the relation of predicate transformer semantics to the Floyd-Hoare Logic as follows:

> *"The way in which we use predicates (as a tool for defining sets of initial or final states) for the definition of the semantics of programming language constructs has been directly inspired by [Charles A.R.] Hoare [39], the main difference being that we have tightened things up a bit: while Hoare introduces sufficient preconditions such that the mechanisms will not produce the wrong result (but may fail to terminate), we shall introduce ... so-called weakest preconditions such that the mechanisms are guaranteed to produce the right results."*

In fact, the Floyd-Hoare Logic is not a completely new development. A lot of ideas on which this logic is based have been published first by Robert W. Floyd in 1967 with the intent to *provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages* [33].

The approach of predicate transformer semantics can be considered as a method to define the semantics of an imperative programming language. This is achieved by so-called predicate transformer which have to be assigned to each language construct. In this context, a predicate transformer is a function which maps from one predicate to another [62].

The canonical predicate transformer of sequential imperative programming languages are the so-called weakest preconditions $WP\ (S, R)$. These define that for a given list of statements S and a condition R on the final state space, the weakest precondition $WP\ (S, R)$ is the weakest condition on the initial state [27]:

> *"... we shall use the notation $WP\ (S, R)$, where S denotes a statement list and R some condition on the state of the system, to denote the weakest precondition for the initial state of the system such that activation of S is guaranteed to lead to a properly terminating activity leaving the system in a final state satisfying the postcondition R. Such a ... [weakest precondition] is called a predicate transformer because it associates the precondition to any postcondition R ..."*

In other words, if S is started in a state satisfying $WP\ (S, R)$, it is guaranteed to terminate in a state satisfying R [67]. The following example shows how the weakest

precondition $WP\ (x = 2 * i + 10,\ 3 * x > 66)$ can be calculated:

$$
\begin{aligned}
&WP\ (x = 2 * i + 10,\ 3 * x > 66)\\
&= \{3 * (2 * i + 10) > 66\}\\
&= \{6 * i + 30 > 66\}\\
&= \{6 * i > 36\}\\
&= \{i > 6\}
\end{aligned}
$$

As mentioned before, the validity of FermaT transformations can be proven with the aid of the predicate transformer semantics approach. To achieve this, it has to be shown that the corresponding weakest precondition of the initial program P is equivalent to the corresponding weakest precondition of the final program P' [96]. For example, to prove the validity of the Reverse Order transformation which is defined as:

$$\Delta \vdash IF\ B\ THEN\ S_1\ ELSE\ S_2\ FI \approx IF\ \neg B\ THEN\ S_2\ ELSE\ S_1\ FI$$

it is necessary to show that the weakest precondition $WP\ (IF\ B\ THEN\ S_1\ ELSE\ S_2\ FI,\ R)$ and the weakest precondition $WP\,(IF\ \neg B\ THEN\ S_2\ ELSE\ S_1\ FI,\ R)$ are equivalent:

$$
\begin{aligned}
&WP\ (IF\ B\ THEN\ S_1\ ELSE\ S_2\ FI,\ R)\\
&= B \Rightarrow WP\,(S_1,\ R) \wedge \neg B \Rightarrow WP\,(S_2,\ R)\\
&= \neg B \Rightarrow WP\,(S_2,\ R) \wedge \neg\,(\neg B) \Rightarrow WP\,(S_1,\ R)\\
&= WP\,(IF\ \neg B\ THEN\ S_2\ ELSE\ S_1\ FI,\ R)
\end{aligned}
$$

Weakest preconditions are the most common predicate transformers because of their relevance to sequential programming. Nevertheless, they are not the only ones. For example, Leslie Lamport has published a paper in 1990 in which he suggested Win and Sin as predicate transformers for concurrent programming [53].

3.2 Wide Spectrum Language

The Wide Spectrum Language (WSL) was developed by Martin Ward at the University of Oxford in the late 1980s. The first ideas of such a language were published by Franz Geiselbrechtinger, Wolfgang Hesse, Bernd Krieg-Brückner and Helge Scheidig in 1973 and 1974 [34, 35]. A refinement of these ideas were published by Friedrich L. Bauer, Manfred Broy, Rupert Gnatz, Wolfgang Hesse, Bernd Krieg-Brückner, Helmut Partsch,

Peter Pepper and Hans Wössner in 1978 [8] whereas the first extensive description of the actual WSL language was published by Martin Ward in 1989 [83].

WSL can be considered as one of the key components of the FTE because all available FermaT transformations are only applicable on programs which are written in this language. The intent of WSL is to provide a reliable transformation language which covers a wide abstraction spectrum. In fact, it is possible to write statements which are similar to Assembler statements next to abstract specifications within one and the same WSL program [96]. According to Martin Ward, there were various requirements which led to the decision to develop another programming language rather than using and extending an existing one [84]. These requirements can be summarised as follows:

1. The language has to provide a simple semantics and tractable reasoning methods which support not only program transformations but also program abstractions and refinements.

2. The implementation of possibly non-executable specifications has to be supported. Furthermore, a mixture of specifications and other statements within one and the same program for stepwise program abstractions and refinements has to be allowed.

3. The FTE and particularly the FermaT transformations have to be independent from a specific programming language. Therefore, an autonomous language has to serve as intermediate language. Accordingly, it has to be possible to translate this autonomous language to other languages and vice versa.

4. The language should be as free as possible from restrictions. This applies particularly to the limitations which are introduced if the language has to be executable. Furthermore, the quirks and foibles of some programming languages which greatly complicate the semantics should be avoided.

To meet these requirements, WSL is based on a so-called kernel language approach [96]. This approach has some major advantages which have been published by Juris Reinfelds in 2002 [70]:

> *"The kernel language approach provides a precise and concise basis for programming in all paradigms (imperative, logical, functional and object-oriented) as well as for parallel, concurrent and distributed multi-thread programming."*

WSL is not based on a functional kernel language. It is based on an imperative kernel language which is extended by so-called functional constructs. These functional constructs are added by means of definitional transformations which define them on the basis of existing constructs [99]. The extensions to the kernel language include practical programming constructs such as action systems, assignments, conditions, functions, loops and procedures [83]. Common programming tactics such as iteration and recursion are supported and even a wide spectrum type system and object-oriented principles have been recently developed to extend WSL [60, 51].

3.2.1 Kernel Language

The kernel language of WSL is based on infinitary first-order logic which was first published by Carol Karp in 1964 [45]. Infinitary first-order logic in turn is an extension of standard first-order logic which allows conjunction and disjunction over countably infinite lists of formula and quantification over finite lists of variables [96].

However, the kernel language consists of only four primitive statements and three compound statements. The state space of this language is defined as a set of variables [95]. Only two primitive statements are able to change the state space at all. This makes the language relatively comprehensible. The four primitive and the three compound statements can be described as follows [90]:

1. Assertion $\{P\}$ is a primitive statement which acts as a partial *SKIP* statement. It terminates if P is true or it aborts if P is false where P can be any infinitary first-order logic formula. The assertion statement does not change the values of any variables.

2. Guard $[Q]$ is a primitive statement which always terminates and enforces Q to be true where Q can be any infinitary first-order logic formula. The guard statement does not change the values of any variables.

3. Add Variables $ADD\ (x)$ is a primitive statement which appends x to the state space if it does not already exist where x can be any finite list of variables. Moreover, it assigns arbitrary values to x which can be restricted by a subsequent guard statement.

4. Remove Variables $REMOVE\ (y)$ is a primitive statement which deletes y from the state space if it exists where y can be any finite list of variables.

5. Sequence $(S_1;\ S_2)$ is a compound statement which executes S_1 followed by S_2 where S_1 and S_2 can be any primitive or compound statement.

6. Nondeterministic Choice $(S_1 \cap S_2)$ is a compound statement which executes either S_1 or S_2 in a nondeterministic manner where S_1 and S_2 can be any primitive or compound statement.

7. Recursion $(\mu X.S)$ is a compound statement which executes S in a recursive manner where S can be any primitive or compound statement and X can be any statement variable which occurs within S. The occurrences of X within S represent recursive calls to S.

Some of these statements may seem quite unusual for many programmers. This applies in particular to the guard statement which is, taken by itself, unimplementable [99]. However, more common statements which are often considered to be atomic can be constructed out of the fundamental statements of the kernel language. A good example is the construction of the assignment $x := 1$ on the basis of a sequence which combines the primitive statement $ADD\ (x)$ with the primitive statement $[x = 1]$:

$$x := 1 \approx ADD\ (x);\ [x = 1]$$

The construction of an assignment which increments the value of the variable x by one requires the storage of the new value in a so-called primed variable x' before it can be copied to the original variable. This can be achieved with the aid of a sequence of primitive statements as well:

$$x := x+1 \approx ADD\ (x');\ [x' = x+1];\ ADD\ (x);\ [x = x'];\ REMOVE\ (x')$$

Another good example is the construction of a condition on the basis of a nondeterministic choice which combines two sequences. Each of these sequences in turn consists of two primitive statements:

$$IF\ B\ THEN\ S_1\ ELSE\ S_2\ FI \approx ([B];\ S_1) \cap ([\neg B];\ S_1)$$

In this context, B can be any statement which returns a boolean whereas S_1 and S_2 can be any lists of statements.

3.2.2 Extensions to the Kernel Language

As mentioned before, WSL is based on an imperative kernel language which is extended by so-called functional constructs. These constructs are added on the basis of the four primitive as well as the three compound statements and are often referred to as the extensions to the kernel language [52]. In fact, the functional constructs are sometimes considered to be the actual WSL whereas the kernel language is often regarded as the theoretical foundation. This is because the kernel language is rarely used for practical tasks such as programming [99].

However, the first extensions have been developed in the late 1980s and have been published by Martin Ward 1989 [83]. Since that time, various new extensions have been added whereas the existing ones have been continuously improved [99, 96]. Recent extensions to the kernel language include various new data types and data structures as well as classes and methods [60, 51].

Although none of the extensions is irrelevant for the proposed approach of this thesis, there are simply too many to describe all of them in this place [84, 87, 95]. Therefore, the following sections provide only a brief overview of the ones which are particularly important in terms of this thesis.

Data Types and Data Structures

The original approach of WSL supports only the data types boolean and integer as well as some data structures like array and list [74]. Furthermore, it does not provide a proper type system which means that the meaning of a variable has to be interpreted. In other words, it is impossible to declare the type of a variable at all. This in turn does not only effect the integrity of WSL programs. It leads also to problems during the translation of such programs to other languages and vice versa [50]. For instance, the data type boolean in WSL is actually an integer where two particular values are used to simulate $TRUE$ and $FALSE$.

To solve these problems, Matthias Ladkau developed a wide spectrum type system for WSL. This system includes not only various additional data types such as short, long and double but also a set of typing rules to provide type safety and therefore to increase the integrity of WSL programs. The first extensive description of this type system has been published in 2009 [51]. The following listing shows an example of a 32 bit signed integer variable in WSL.

Listing 3.3: WSL Example Program to Demonstrate a 32 Bit Signed Integer Variable.

```
1 VAR <INTEGER*4 :: i := 0>:
2     i := i + 1
3 ENDVAR
```

However, the development of the wide spectrum type system has been carried out at the same time as the development of the constraint based program transformation system. Furthermore, there exist only a prototype tool of this type system which has a lot of limitations. For these reasons, the examples and the case studies which are presented in this thesis have been created on the basis of the original WSL approach.

Variables

The original approach of WSL supports local and global variables. Local variables have to be declared within a local block before they can be used. This local block is also referred to as variable block. Global variables in turn cannot be explicitly declared at all. Instead, they are automatically declared where they first appear within a WSL program. The following listing shows an example program which contains a local variable *i* and a global variable *j*.

Listing 3.4: WSL Example Program to Demonstrate a Local and a Global Variable.

```
1 VAR <i := 0>:
2     j := i
3 ENDVAR;
4 PRINT(j)
```

The recent introduction of the wide spectrum type system has changed the handling of global variables in WSL. If this system is used, global variables have to be declared similar to local variables [51].

Conditions and Guarded Commands

WSL supports common conditions as well as so-called guarded commands. On the one hand, a common condition is one which appears in many other programming languages such as C, Cobol and Java as well. The syntax might vary in some cases but the structure is always as follows:

$$IF\ B\ THEN\ S\ FI$$

or

$$IF\ B\ THEN\ S_1\ ELSE\ S_2\ FI$$

In this context, B can be any statement which returns a boolean whereas S, S_1 and S_2 can be any lists of statements. There is also an *ELSIF* statement which allows structures that are commonly known as *CASE* structures. On the other hand, a guarded command is one which appears to be rather unusual. Martin Ward described the function of guarded commands in terms of WSL as follows [93]:

> *"All the conditions [of the guarded command] are evaluated. If any condition is true then one of the statements corresponding to a true condition is selected for execution in a nondeterministic way. If none of the conditions evaluates to true then the statement terminates. Note that with a guarded command, the order of the clauses is irrelevant."*

The idea of guarded commands was first published by Edsger W. Dijkstra in 1975 [27]. A more extensive descriptions was published in 1976 [28]. The following listing shows an example of a guarded command written in WSL.

Listing 3.5: WSL Example Program to Demonstrate a Guarded Command.

```
1 D_IF EVEN?(i) ->
2    j := 1
3 [] ODD?(i) ->
4    j := 2
5 FI
```

Loops

WSL supports various kinds of loops. This includes the common *FOR* and *WHILE* loops as well as *DO* loops and even guarded loops. *FOR* and *WHILE* loops appear in many other programming languages such as C, Cobol and Java as well. These loops belong to the group of bounded loops because their termination depends on a condition which is part of the loop construct itself. The syntax might vary in some cases but the structure is always as follows:

FOR i TO n STEP j DO S OD

and

WHILE B DO S OD

In this context, i, j and n are integer variables, B can be any statement which returns a boolean and S can be any list of statements. WSL supports also loops which do not provide a termination condition. These loops belong to the group of unbounded loops and are referred to as *DO* loops. The termination of these loops depends on an $EXIT\ (n)$ statement which causes the program to exit the n enclosing *DO* loops [93]. Guarded loops in turn are rather unusual. The function of guarded loops in terms of WSL is similar to the function of guarded commands but with the difference that the statements which are corresponding to a true condition will be executed as long as there are any true conditions. The idea of guarded loops was first published by Edsger W. Dijkstra in 1975 [27]. A more extensive descriptions was published in 1976 [28]. The following listing shows an example of a guarded loop written in WSL.

Listing 3.6: WSL Example Program to Demonstrate a Guarded Loop.

```
1 D_DO EVEN?(i) AND i < 100 ->
2   i := i + 1;
3   j := j + 1
4 [] ODD?(i) ->
5   i := i + 1;
6   k := k + 1
7 OD
```

As mentioned before, the function of guarded loops in terms of WSL is similar to the function of guarded commands. Therefore, it is possible to transform a guarded loop into a common *DO* loop which contains a guarded command. For example, this can be achieved with the aid of the Dijkstra Do to Floop transformation. The following listing shows a program which is the result of this FermaT transformation applied on the program which has been shown in the previous listing.

Listing 3.7: WSL Example Program to Demonstrate a *DO* Loop.

```
1  DO
2    D_IF EVEN?(i) AND i < 100 ->
3      i := i + 1;
4      j := j + 1
5    [] ODD?(i) ->
6      i := i + 1;
7      k := k + 1
8    [] i >= 100 AND EVEN?(i) ->
9      EXIT(1)
10   FI
11 OD
```

Specifications

As mentioned before, the abstraction and the refinement of a program is one of the main tasks of the FTE. To perform these task with the aid of automated program transformation processes is very important for reverse and forward engineering [99]. For example, it has many advantages to abstract an executable program into a specification or to refine a specification into an executable program. Since WSL is one of the key components of the FTE, it supports abstract specification statements which describe *what* a program does without describing *how* it is done.

An individual WSL specification statement is not more than a list of variables and a formula which describes the relation between the old and the new values of these variables. The general notation of such a statement is $x := x'.Q$ where x is a list of variables, x' is the corresponding list of so-called primed variables and Q is any formula [95]. The formula Q specifies the program whereas the list of variables x contains the initial values

and the list of primed variables x' contains the final values. For example, the statement

$$\langle i \rangle := \langle i' \rangle \,.\, (i' = i + 1)$$

specifies a program which increments the value of the variable i by one. The arrow brackets indicate a list of variables which means that the list x contains only the variable i. Accordingly, the list x' contains only the primed variable i'. The following listing shows how this specification statement appears within a WSL program.

Listing 3.8: WSL Example Program to Demonstrate a Specification Statement.

```
1 SPEC  <i>:
2     i ' = i + 1
3 ENDSPEC
```

Action Systems

An action system is a collection of parameterless procedures which are called actions [74]. One of these actions is the so-called initial action which will be executed after entering the system. The name of this initial action has to be stated in the first line of the system after the keyword *ACTION*. The purpose of action systems is to simplify the translation from Assembler languages to WSL [93]:

> *"A program written using labels and jumps ... [can be converted] directly into an action system by translating each labelled block of code as an action and each GOTO statement as an action call. Where one block of code falls through to the next, we add an explicit action call."*

An individual action can contain various statements. If all of them have been executed, the program will continue at the position where the action was called. The action system terminates only if there are no statements left which have to be executed. An exception is the special action call Z which terminates the action system immediately. For this reason, Z must not be used as name of an action. If the execution of the action system has been terminated, the program carries on with the statement which comes after the action system. The following listing shows an example program which contains an action system. The initial action of this system is the one with the name *PROG*.

Listing 3.9: WSL Example Program to Demonstrate an Action System.

```
1  ACTIONS PROG:
2    PROG ==
3      i := 0;
4      CALL A
5    END
6    A ==
7      j := 1;
8      CALL B
9    END
10   B ==
11     k := 2;
12     CALL C
13   END
14   C ==
15     l := 3;
16     CALL Z
17   END
18 ENDACTIONS
```

Procedures and Functions

WSL supports common procedures as well as functions and so-called boolean functions [52]. On the one hand, a common procedure is a definition which appears in many other programming languages such as C and Java as well. Procedures in C are commonly called functions whereas procedures in Java are commonly called methods [43, 77]. The syntax might vary in some cases but the structure is always as follows:

$$PROC\ P\,(v_1,\ v_2,\ \ldots\ v_n\ VAR\ r_1,\ r_2,\ \ldots\ r_n) ==\ S\ END$$

In this context, the variables $v_1,\ v_2,\ \ldots\ v_n$ are so-called initialisation parameters whereas the variables $r_1,\ r_2,\ \ldots\ r_n$ are so-called result parameters. A slight difference of WSL procedures compared to C functions or Java methods is that a WSL procedure can return a number of variables whereas a C function and a Java method can return only one vari-

able. On the other hand, WSL functions which start with the keywords *FUNCT* and WSL boolean functions which start with the keyword *BFUNCT* are definitions which appear to be rather unusual. These functions do not provide an ordinary list of statements in their body but a local variable block and either a single condition or another statement which provides a single integer or boolean value surrounded by brackets [74]. The following listing shows an example of a function written in WSL.

Listing 3.10: WSL Example Program to Demonstrate a Function.

```
1 FUNCT F(i, j) == VAR <k := 0, l := 1>:
2   (IF i > 0 THEN
3     k
4   ELSE
5     l
6   FI)
7 END
```

Procedures as well as functions and boolean functions in WSL have to be surrounded by a *WHERE* clause. Therefore, they have boundaries similar to local variables. Functions return always a single integer value whereas boolean functions return always a single boolean value. Furthermore, WSL allows references to external procedures, functions and boolean functions where external means that they are implemented in other languages [52].

3.3 Implementation of Transformations

As mentioned before, an individual transformation can be defined as an operation on an AST. This applies not only in regard to the FTE but also in view of other program transformation approaches like DMS or Stratego/XT. Furthermore, each of these approaches provides some tools to simplify the development of new transformations.

In terms of the FTE, there are three tools which are basically implementing a powerful transformation description language called $\mathcal{META}$WSL. All FermaT transformations are written in this language. $\mathcal{META}$WSL is an extension of WSL which has the advantage that the FermaT transformations can be applied on their own source code [52]. This can be helpful in order to improve the implementation of such transformations. Martin Ward

and Hussein Zedan described $\mathcal{META}$WSL in 2005 as follows [94]:

> *"[For the implementation of the FermaT Transformation Engine] we decided to extend WSL to add domain-specific constructs, creating a language for writing ... [FermaT] transformations. This [language] was called* $\mathcal{META}$WSL. *The extensions include an abstract data type for representing programs as tree structure and constructs for pattern matching, pattern filling and iterating over components of a program structure."*

The three tools which are implementing $\mathcal{META}$WSL are a $\mathcal{META}$WSL to Scheme translator, a Scheme runtime library and a $\mathcal{META}$WSL runtime library. Scheme is a multi-paradigm programming language and a dialect of Lisp. It was developed by Guy L. Steele and Gerald J. Sussman in the 1970s as a language which provides an exceptionally clear and simple semantics [46]:

> *"[Scheme] was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional and message passing styles, find convenient expression in Scheme."*

According to Martin Ward and Hussein Zedan, Scheme has been selected as translation target because it is a *small, well-defined language which has an ISO and an ANSI standard and which has good facilities for manipulating trees and lists* [94]. However, the translation of $\mathcal{META}$WSL programs to Scheme is an automated and therefore quite simple process. The essential commands of $\mathcal{META}$WSL are implemented in the Scheme runtime library whereas the complex commands of this language are implemented in the $\mathcal{META}$WSL runtime library.

3.3.1 Essential Commands

The Scheme runtime library provides the essential commands of $\mathcal{META}$WSL which can be identified by the prefix @. These commands in turn are split into four groups which are the basic commands, the navigation commands, the getter commands and the editing commands. The following sections provide an overview of the four groups.

Basic Commands

$\mathcal{META}$WSL supports several dozens of basic commands which are too many to describe all of them in this place [74]. Therefore, some relatively common and important basic commands have been selected which are as follows:

1. *@Make*(v, l) creates an AST type with the value v and the list of children l. The parameter v is required to create some particular AST types whereas the parameter l is optional.
2. *@Parse* creates an AST of the program which is loaded into the FTE.
3. *@Pass* terminates the enclosing procedure and returns true.
4. *@Fail*(m) terminates the enclosing procedure and returns false and the message m. Does not reverse the changes which have been performed by the procedure.
5. *@McCabe* returns the cyclomatic complexity of the current program.
6. *@NewProgram*(f) loads the program which is located within the file f into the FTE. Returns an error message if the file could not be found or if there are any syntax errors within the program.
7. *@Variables* returns a set which contains all variables of the current program.

Navigation Commands

As discussed in Chapter 2, an AST of a WSL program contains AST types as nodes and leafs. Each of these types is either a general type, a group type or a specific type and is nested according to the rules of WSL. These rules will be described in Appendix A.

To navigate through the AST of the current program, the Scheme runtime library stores the selected AST path where so-called navigation commands can be used to alter the selection. In terms of $\mathcal{META}$WSL commands, the current program is the one which has been parsed at last. The selected AST path in turn can be considered as sequence of AST types which have to be visited to reach the target type. The most common and important navigation commands are as follows [52]:

1. *@Up* selects the parent of the currently selected AST type if it exists. The existence can be checked with the *@Up*? command.

2. *@Down* selects the first child of the currently selected AST type if it exists. The existence can be checked with the *@Down*? command.

3. *@Left* selects the left neighbour of the currently selected AST type if it exists. The existence can be checked with the *@Left*? command.

4. *@Right* selects the right neighbour of the currently selected AST type if it exists. The existence can be checked with the *@Right*? command.

5. *@Goto*(p) selects the AST type which has the AST path p if it exists. The existence can be checked with the *@Goto*?(p) command.

6. *@Posn* returns the AST path of the currently selected AST type.

Getter Commands

There are also a number of commands which return the entire AST of the current program or just a specific part of it. These commands are often referred to as getter commands. The most common and important getter commands are as follows [94]:

1. *@Program* returns the entire AST of the current program.

2. *@Item* returns the currently selected AST type.

3. *@Type*(t) returns the particular general type, group type or specific type of the AST type t.

4. *@Parent* returns the parent of the currently selected AST type if it exists.

5. *@Value* returns the value of the currently selected AST type if it has a value.

6. *@Buffer* returns the content of the Scheme runtime library buffer. This command is required to access AST types which have been cut out by the *@Cut* command.

Editing Commands

Furthermore, there are so-called editing commands which can be used to change the AST and with it the current program itself. These commands are responsible for any editing which will be performed by the FermaT transformations. The most common and important editing commands are as follows [74]:

1. *@Delete* deletes the currently selected AST type without checking the syntactic correctness of the resulting program.

2. *@CleverDelete* deletes the currently selected AST type only if the resulting program is syntactically correct.

3. *@Cut* copies the currently selected AST type into the Scheme runtime library buffer and applies the *@Delete* command on the type. The buffer can be accessed by the *@Buffer* command.

4. *@PasteOver*(t) replaces the currently selected AST type by the AST type t.

5. *@PasteBefore*(t) inserts the AST type t before the currently selected AST type.

6. *@PasteAfter*(t) inserts the AST type t after the currently selected AST type.

7. *@Rename*(t) changes the name of the currently selected AST type if it has a name.

3.3.2 Complex Commands

The $\mathcal{META}$WSL runtime library provides the complex commands of this language. The commands do not have a particular prefix but they are written in capital letters. $\mathcal{META}$WSL supports various complex commands which can be used in different ways. There are simply too many to describe all of them in this place [74, 49]. Therefore, some relatively common and important complex commands have been selected which are as follows:

1. *ERROR* aborts the application of a FermaT transformation and reverses all the changes which have been performed by this transformation.

2. *FOREACH* is a loop which iterates over all general types, group types and specific types of a defined AST where a particular AST type has to be selected to indicate the root of the defined AST.

3. *FILL* returns the AST of a WSL statement.

4. *IFMATCH* performs a pattern match on a defined AST where a particular AST type has to be selected to indicate the root of the defined AST.

5. *MEMBER*? returns if a particular general type, group type or specific type occurs within a defined AST where a particular AST type has to be selected to indicate the root of the defined AST.

6. *MW_PROC* indicates a specific $\mathcal{M}_{ETA}$WSL procedure which behaves similar to an ordinary WSL procedure but with the difference that it does not have to be surrounded by a *WHERE* clause.

3.3.3 Design of FermaT Transformations

In general, the source code of a FermaT transformation is split into two $\mathcal{M}_{ETA}$WSL procedures. The first procedure is the applicability condition which checks if the transformation is applicable whereas the second procedure is the transformation definition which describes how the FermaT transformation changes the program. The following listing shows the source code of the Delete All Skips transformation.

Listing 3.11: $\mathcal{M}_{ETA}$WSL Code of the Delete All Skips Transformation.

```
1 MW_PROC @Delete_All_Skips_Test() ==
2   IF MEMBER?(T_Skip, @Type(@Item)) THEN
3     @Pass
4   ELSE
5     @Fail("test: no SKIP statement found")
6   FI
7 END;
8 MW_PROC @Delete_All_Skips_Code() ==
9   FOREACH Statement DO
10    IF @Type(@Item) = T_Skip THEN
11      @Delete
12    FI
13  OD
14 END
```

The source code is split into the *@Delete_All_Skips_Test*() procedure which is the applicability condition and the *@Delete_All_Skips_Code*() procedure which is the transformation definition. In addition to standard WSL statements, the applicability condition uses the essential and the complex commands of $\mathcal{M}_{ETA}$WSL to check if the selected AST

type which indicates the root of a defined AST contains the specific type T_Skip. If this is the case, it uses the *@Pass* command to terminate the procedure and to return true. Otherwise, it uses the $@Fail\,(m)$ command to terminate the procedure and to return false and an error message. The transformation definition in turn uses i.a. the *FOREACH* loop to delete all *SKIP* statements within this defined AST.

3.4 Summary

The presented chapter has discussed the theoretical foundation of the FermaT Transformation Engine (FTE). It has provided an introduction to program transformation theory and explained the basic principles of the Wide Spectrum Language (WSL). Furthermore, it described the implementation of FermaT transformations on the basis of a transformation description language which is called $\mathcal{META}$WSL.

Chapter 4

Definition and Classification of Constraints

Objectives

- To define and classify constraints in the area of program transformation theory.
- To observe program transformation processes from different perspectives.
- To discuss the connection of constraints to program characteristics and properties.
- To explain the dependence of constraints to various measuring techniques.

This chapter defines program transformation constraints, introduces a classification of these constraints and discusses their importance and their benefit for a successful use of program transformation theory. It observes program transformation processes from different perspectives and discusses the relation of constraints to program characteristics and properties. Furthermore, the dependence of constraints to various measuring techniques such as pattern matching and metrics are presented and explained.

The general aim of constraints is to define program transformation targets. After their definition, these targets should be achieved in a totally automated manner through the application of a transformation sequence. Therefore, the search for such sequences will

be extensively discussed in Chapter 7. However, some of the defined constraints are currently only achievable with human interaction.

The different groups of constraints within the provided classification can also be considered as different perspectives to look at the maintenance of programs. They are very strong connected to each other and overlap partially.

4.1 Definition of Program Transformation Constraints

In terms of this thesis, the program P_0 is the initial program which has to be transformed. The program P_n is the final program which is wanted. A constraint C_j is defined as a condition which may have to be satisfied during a program transformation process. There can be any number of constraints included in an individual program transformation process where each constraint C_j has to be assigned to a particular state S_i within a model of this process. These models are called transformation schemes whereas the included states $S_0 \ldots S_n$ are called scheme states which will be discussed in Chapter 6. Once this model has been developed, it is possible to automatically create a set of transformation sequences Ω which are defined by the model. This will be discussed in Chapter 7.

Each transformation sequence T_{Si} within the created set Ω is unique. They all consists of FermaT transformations and constraints which have been transferred from the transformation scheme. Moreover, a sequence is only valid if each included transformation $t_0 \ldots t_n$ is applicable and if each included constraint $C_1 \ldots C_m$ has been satisfied. The included constraints in turn are assigned to a particular program state P_i. These program states only appear within transformation sequences and should not be confused with scheme states. Due to the assignment, each constraint C_j has to be satisfied by the respective program state P_i:

$$\forall j : P_i \text{ sat } C_j$$

To prove the validity of a transformation sequence T_{Si}, the applicability of each transformation t_i within this sequence has to be checked. This requires that each transformation except the last has to be applied. Moreover, the satisfaction of each constraint C_j within the sequence has to be checked which requires that the corresponding program state P_i

has to exist. This means that even the last transformation t_{n-1} of the sequence has to be applied if there are constraints assigned to the final program P_n. The application of a transformation sequence can be considered as follows:

$$P_0\ t_0\ P_1\ t_1\ P_2\ t_2\ ...\ t_{n-1}\ P_n$$

The only program state which exists at the beginning is the inital program P_0. Accordingly, the application of a transformation t_i creates a new program state P_{i+1} on which the next transformation t_{i+1} has to be applied. The program state which has been created by the last transformation t_{n-1} of a transformation sequence T_{Si} is the final program P_n. Therefore, a transformation sequence can be described as follows:

$$T_{Si} = t_n\left(t_{n-1}\left(t_{n-2}\left(...t_0\left(P_0\right)...\right)\right)\right)$$

If a transformation sequence T_{Si} turns out to be invalid, it will be discarded and the next sequence T_{Si+1} within the created set of transformation sequences Ω will be applied. Since such a set can be quite extensive, it is often difficult to find a valid sequence. If there exists no valid sequence within the set Ω, the program transformation process has failed. If this is the case because none of the constraints could be satisfied, the used transformation scheme can be considered as overconstrained. The contrary is that each sequence T_{Si} within the set Ω satisfies all included constraints. In this case, the used transformation scheme can be considered as underconstrained. Both cases usually indicate some errors or false assumptions during the transformation scheme development and are not desirable at all.

As mentioned before, a constraint can be considered as condition. Therefore, it can only be satisfied or not satisfied. However, it is possible to combine various constraints where each of them has to be satisfied. There exist two different main classes of constraints:

1. **Structural Constraints**
   The first class comprises structural constraints which regard the static representation of a program. This is the source code within the scope of this thesis. For this reason, structural constraints are defined to address an individual characteristic or a set of characteristics. A characteristic in turn is defined to be a structural element of a

program. For instance, a characteristic can be a statement within the source code because a statement is a structural element. Furthermore, it can be a specific pattern or the amount of a particular statement. The maximum nesting depth of a program is a characteristic as well as the cyclomatic complexity. The LoC and the NoCC of a program are also characteristic because they all regard its source code. Even the programming paradigm is a characteristic.

However, the satisfaction of a structural constraint is checked by inspection. For example, a constraint can be used to define that the cyclomatic complexity of a program state P_i has to be less than 15. In this case, a technique is required to measures the cyclomatic complexity of the particular state.

2. **Behavioural Constraints**

   The second class comprises behavioural constraints which regard the application of a program. In contrast to structural constraints, behavioural constraints are defined to address an individual property or a set of properties. A property in turn is defined to be a behavioural element of a program. For instance, a property can be the execution time of a program which runs on a particular system. Furthermore, it can be the memory consumption of a program while it is executed. The actual complexity of a program can also be a property because it can be measured on the basis of a technique which counts the conditional branches which are used during a case of application.

   However, a behavioural constraint is satisfied if the addressed property is an element of the set of program properties or if the set of addressed properties is a subset of the set of program properties. For example, a constraint can be used to define that the execution time of a program state P_i has to be less than five seconds. In this case, a technique is required to measure the execution time of the particular state.

As mentioned before, a characteristic always regards the structure of a program whereas a property always regards its behaviour. The structure in turn has a significant influence on the behaviour of a program. There are some cases where the only difference between a characteristic and a property is the context in which they are considered. For example, a condition is a characteristic because it is a structural element. It consists of various characters and increases the cyclomatic complexity of a program [63]. On the other hand, the same condition can be compiled into an executable form where it may cause a particular behaviour during the program execution. This behaviour may not only affect the final

states in which the program terminates. It may also affect a lot of other properties like the execution time or the memory consumption of the entire program. For this reason, the same condition can also be considered as behavioural element and therefore as property.

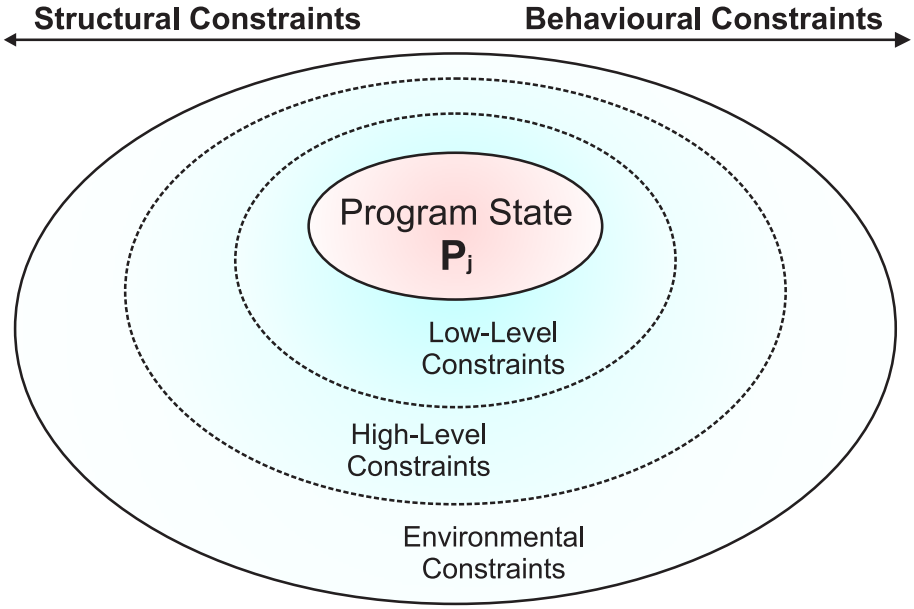
One of the weak points of the program property approach is the gap between the high-level source code and the generated executable of a program. Some statements of the high-level code might be deleted or modified during the compilation process. For example, a comment is a statement which will not be transferred to the executable whereas the appearance of conditions and loops will probably be changed. Furthermore, the optimisations which are carried out by modern compilers are often unpredictable for a maintainer. Therefore, behavioural constraints which are based on program properties have to be defined carefully.

In many cases, it is obvious whether a particular constraint belongs to the class of structural or behavioural constraints. This makes the classification of them very comprehensible. However, there are constraints which seem to belong to both classes. This applies especially for the high-level and environmental constraints which will be discussed in the course of the next paragraphs. Furthermore, structural constraints have often a critical influence on some of the behavioural constraints and vice versa. For instance, a structural constraint on a time-critical loop can have an impact on several different behavioural constraints. On the other hand, it cannot be ignored that FermaT transformations are only able to transform source code written in WSL but not the executable code. Therefore, we have to change the source code to achieve behavioural constraints which will possibly have an impact on some structural constraints.

Constraints can not only be classified into structural and behavioural constraints but also into low-level, high-level and environmental constraints. The borders between these different classes of constraints are very soft. There are three main classification criteria:

1. **Range of Effect**
   The first criterion is that low-level constraints are affecting only a small fraction of a program whereas high-level and environmental constraints affect the majority or even the entire program. This criterion is caused by the existence of local characteristics and properties as well as global characteristics and properties. A local

Figure 4.1: Classification of Constraints for Program Transformation.



characteristic of a program covers only a fraction of its structure where it is required that the position of this fraction has been specified by any means. This is beneficial because it is not necessary to check the entire program to determine whether a constraint has been satisfied or not. The same applies to a local property of a program which covers only a part of its application. On the other hand, a global characteristic covers the majority or the entire structure whereas a global property covers the majority or the entire application of a program.

A suitable example for low-level constraints in view of this classification criterion are pattern constraints. These can be used to define that a particular structure has to exist at an explicit position within a program. In this case, the involved pattern is a characteristic which covers only a fraction of an entire program. It is only necessary to check this fraction to determine whether the constraint is satisfied or not. This makes the pattern a local characteristic which means that the corresponding constraint belongs to the group of low-level constraints. The classification gets more difficult if a pattern constraint is used to define that a particular structure has to exist somewhere within the program. In this case, it might be necessary to check the entire program for this pattern. Nevertheless, it is still a characteristic which covers only a fraction of a program. This is because it is possible to determine the posi-

tion of the pattern within the structure once it has been found. In other words, the pattern which causes a satisfaction of a pattern constraint has an explicit position within the structure of a program even if this is not known. Therefore, the pattern can still be considered as local characteristic and the corresponding constraints as low-level constraint.

However, pattern constraints can indeed belong to the group of high-level constraints. This is the case if the involved pattern covers the majority or even the entire program. A suitable example for high-level constraints in view of this classification criterion are metric constraints. These can be used to restrict the cyclomatic complexity of a program. In general, a metric covers the entire program and not just a little fraction of it. Additionally, it is not possible to determine an explicit position which causes the satisfaction of a high-level constraint. Therefore, the cyclomatic complexity of a program can be considered as global characteristic which means that the corresponding constraint belongs either to the group of high-level constraints or to the group of environmental constraints.

2. **Hierarchy**

   The second classification criterion concerns the constraint hierarchy. On the one hand, an individual high-level or environmental constraint can consist of one or more low-level constraints. On the other hand, a low-level constraint must not depend on other constraints.

   A suitable example for environmental constraints in view of this classification criterion are compiler constraints. These can be used to describe the restrictions of various compilers and compiler versions for different platforms. A particular compiler constraint can be developed on the basis of a set of low-level constraints which define the prohibited code characteristics.

3. **Intention**

   The third classification criterion distinguishes high-level and environmental constraints. A constraint can be considered as an environmental constraint if it is determined by the environment in which the program is used. This includes the development environment as well as the environment in which the program is supposed to run.

   A suitable example for the distinction of high-level and environmental constraints in view of this classification criterion are abstraction level and programming language

constraints. An abstraction level constraint can be considered as programming language constraint and therefore as environmental constraint if it has been defined in view of the abstraction level which is supported by a particular programming language.

The application of transformations on a given program to satisfy low-level constraints might have an influence on some high-level or environment constraints. On the other hand, the application of transformations on a given program to satisfy high-level constraints or environment constraints will certainly have an influence on some low-level constraints even if they are not important for the current program transformation process. Furthermore, there can be interdependencies between high-level and environment constraints if transformations are applied to achieve them.

4.2 Low-Level (Structural) Constraints

Low-level (structural) constraints address atomic, local characteristics of a source code. As mentioned before, they are checked by inspection. In contrast to FermaT transformations which describe the conversion from one program into another, they are the fundamental basis to describe the targets which have to be achieved during the program transformation process. All other constraints are depending on them in one way or another. Due to the fact that FermaT transformations are only transforming WSL code, even behavioural constraints need to rely directly or indirectly on them.

4.2.1 Pattern Constraints

In the area of computer science, the task of pattern matching is to check for the presence of a defined structure. Pattern constraints are using different pattern matching techniques to determine if a requested target has been achieved or not. By doing so, it is possible to determine whether to achieve or to avoid a particular pattern within the source code of a WSL program. This is a simple but crucial element of constraint based program transformation. Pattern constraints are used as a basis for a lot of other constraints which are far more complex.

Patterns can be defined in different ways. It is possible to implement them directly in a specific programming language. The advantage of this approach is the flexibility

and the huge collection of libraries and tools which can be used. On the other hand, the disadvantage is that a pattern definition would probably be hard to comprehend and its implementation is depending on the chosen language. It is also possible to implement them in a particular tree description language like $\mathcal{META}$WSL which has been developed to define FermaT transformations on a WSL AST. $\mathcal{META}$WSL has been discussed in Chapter 3. However, this language is very complex and it does not even support quantifier or wildcards within the structure. Therefore, a very simple formal language called PDL has been developed. This language has been adapted to the special needs of pattern matching in combination with WSL.

The structure of this language is similar to a WSL AST. It defines a pattern as a tree which contains AST types as nodes and leafs. These types can be general types, group types or specific types and can be nested according to the rules of the WSL language which are described in Appendix A. Furthermore, it is possible to insert quantifier and wildcards at various positions to determine if a particular type appears multiple times or if there are types accepted which are unknown. The following table shows a description of PDL in the Backus-Naur-Form.

Table 4.1: PDL Description in the Backus-Naur-Form.

BNF Type	**Definition**
<*pattern*>	(<not>)? (<contains>)? <ast_type> (<operator> <ast_type>)* ";"
<*not*>	"!"
<*contains*>	"#"
<*ast_type*>	<wsl_ast_type> "{" (<ast_subtype>)* "}" ("[" <quantifier> "]")?
	Continued on next page

PDL Description - continued from previous page.

BNF Type	**Definition**
<operator>	"|" | "||"
<wsl_ast_type>	GENERAL TYPE | GROUP TYPE | SPECIFIC TYPE
<ast_subtype>	<pattern> | (<wildcard> ("[" <quantifier> "]")?)
<quantifier>	NUMBER | "?" | "+" | "*"
<wildcard>	"%"
NUMBER	a natural number
GENERAL TYPE	a general type of the WSL language
GROUP TYPE	a group type of the WSL language
SPECIFIC TYPE	a specific type of the WSL language
";"	a semicolon to indicate the end of a pattern
"!"	a logical negation
"#"	an operator to determine if the pattern at this place contains the following AST type
"{"	a left curly bracket
"}"	a right curly bracket
"["	a left square bracket
"]"	a right square bracket
"|"	a logical disjunction
"||"	an exclusive disjunction
"?"	a quantifier (0 or 1)
"+"	a quantifier (1 to n)
""*	a quantifier (0 to n)
"%"	a wildcard for any AST type of the WSL language

As the Backus-Naur-Form shows, a *pattern* is a non-terminal which consists of an optional *not*, an optional *contains*, an *ast_type* and a number of *operators* each followed by another *ast_type*. An *ast_type* in turn is a non-terminal which consists of a *wsl_ast_type*, a number of *ast_subtypes* which are surrounded by curly brackets and an optional *quantifier* which is surrounded by square brackets. The non-terminal *wsl_ast_type* consists of terminals which represents the general types, group types and specific types of the WSL

language. An *ast_subtype* is either a *pattern* or a *wildcard* followed by an optional *quantifier*. All other non-terminals only consist of terminals and signs. The following listing shows a concrete example of a pattern definition written in PDL.

Listing 4.1: Pattern Definition Example in PDL.

```
1  T_Statements {
2    T_Floop {
3      T_Statements {
4        % [*]
5        T_Assignment {
6          T_Assign {
7            T_Var_Lvalue [1];
8            T_Number [1];
9          } [1];
10       } [1];
11       % [*]
12     } [1];
13   } [1];
14 } [1];
```

The example defines a pattern where the general type T_Assign is nested within four other AST types which are the specific type T_Assignment, the group type T_Statements, the specific type T_Floop and another group type T_Statements. On the other hand, the T_Assign contains the specific types T_Var_Lvalue and T_Number. As defined in the language, each of the nested AST types can be for their part considered as subtypes. Above and below the T_Assignment, there are wildcards with quantifiers as arguments which determine that at these positions there can be a number of any statements. The following listing shows a WSL example program which matches the pattern definition.

Listing 4.2: Pattern Matching Example in WSL.

```
1 DO
2   i := 0;
3   j := j + 1;
4   IF j > 10 THEN
5     EXIT(1)
6   FI
```

```
7 OD
```

For instance, if a pattern constraint is defined to avoid this pattern, a simple transformation sequence exists which can be applied on the WSL example program to satisfy this constraint. An application of the Take Out Left transformation on the specific type T_Assignment followed by an application of the Constant Propagation transformation on the specific type T_Floop will transform the program into another program which is shown in the following listing. In terms of a constraint based program transformation process, this program can be considered as the final program P_n which satisfies the defined pattern constraint. Accordingly, the program which is shown in Listing 4.2 can be considered as the initial program P_0.

Listing 4.3: Pattern Constraint Satisfaction Example in WSL.

```
1 i := 0;
2 DO
3   j := j + 1;
4   IF j > 10 THEN
5     EXIT(1)
6   FI
7 OD
```

Although only primitive patterns have been used at the current stage, there is certainly a lot more potential for pattern in relation with constraint based program transformation. For example, it is imaginable to use design pattern in association with future research projects.

4.2.2 Convention Constraints

Convention constraints are used in correlation with atomic code conventions. These are usually not enforced by the environment but defined and checked by the programmer while the program is written. The ulterior motive of these constraints is often to ensure that the source code of a program is easy to read, mnemonic and consistent. A well known class of conventions are the structural conventions. These include the limitation of the nesting depth, the determination where variables have to be declared or the avoidance of particular AST types within the source code. There are a lot more structural conventions for each programming language which are also depending on the requirements of

the respective project.

The adherence to conventions is a serious problem particularly in reference to automatically generated source code as is the case with program transformation processes. To take the previous structural convention example up again, the resulting code is often massively nested, variable declarations are spread all over the program and there can be any AST types. These problems can be solved through the introduction of constraints into the program transformation process. Currently, three different convention constraints have been defined which can be summarised as follows:

1. **Nesting Depth**
   The first constraint addresses the convention which restricts the maximum nesting depth of statements within a program. This constraint has been defined with the aid of a depth-first-search on the AST of the source code. Therefore, the nesting structures like for example *DO* loops and *IF* statements have been identified and need to be counted for each path which is traversed during the depth-first-search. The following listing shows a WSL example program which contains a *DO* loop and an *IF* statement.

   Listing 4.4: Convention Constraint Example in WSL.

```
1 VAR <i := 0>:
2   DO
3     i := i + 1;
4     IF i > 10 THEN
5       EXIT(1)
6     FI
7   OD
8 ENDVAR
```

   The presented program consists of 26 AST types. The root node of the AST is as always the group type T_Statements which is described in Appendix A. It only contains one T_Var as subtype which is the beginning of a variable block and surrounds other statements. Therefore, the nesting depth for this path through the AST has to be increased by one during the depth-first-search. The body of the local block is represented by another T_Statements. This only contains a *DO* loop which appears

in the AST as specific type T_Floop. Again, other statements are surrounded by the loop and the nesting depth for this path must be increased by one. This technique continues through the entire AST. The path with the highest nesting depth is the one which ends with the specific type T_Exit. It is now possible to determine if this result is within the given limit which means that the convention constraint has been satisfied. The implementation of this algorithm is relatively complex compared to other algorithms which are used in combination with convention constraints. Moreover, it describes a particular source code characteristic as numeric value. Therefore, it could also be considered as metric constraint which will be discussed in one of the following sections.

2. **Variable Declaration and Initialisation**
   The second constraint addresses the convention which regards the declaration and the initialisation of variables. Various conventions determine that variables have to be declared and initialised within the same statement. Furthermore, it is often specified that global variables have to be declared and initialised at the beginning of a module or a class and local variables have to be declared and initialised at the beginning of a local block. To comply with this convention is a serious problem in terms of the FTE because the original approach of WSL does not support a type system [51]. Therefore, it is not possible to declare variables in the current version of the FTE at all. This in turn complicates the identification of variable initialisations. The following listing shows a statement of this kind written in C.

Listing 4.5: Variable Declaration and Initialisation in C.

```
1 int i = 0;
```

In this case, it is easy to identify the declaration and initialisation of the variable *i* because of the keyword *int*. The same statement looks similar in many other languages. However, the following listing shows how the corresponding statement looks in the original approach of WSL.

Listing 4.6: Variable Declaration and Initialisation in WSL.

```
1 i := 0;
```

As mentioned before, the variable declaration is missing. For this reason, it is impossible to distinguish between variable initialisations and simple assignments

which can appear everywhere in the program. A solution for this problem is to define that a WSL program always has to begin with a variable block. Accordingly, each variable which is used within such a program has to be initialised in the head of this block. The corresponding constraint can be implemented on the basis of a pattern description. This can be defined with the aid of the PDL language. The following listing shows an example of a pattern which defines that a WSL program has to begin with a variable block.

Listing 4.7: Convention Constraint Definition Example in PDL.

```
1  T_Statements {
2    T_Var {
3      T_Assigns {
4        T_Assign [*];
5      } [1];
6      T_Statements {
7        % [*]
8      } [1];
9    } [1];
10 } [1];
```

The defined pattern begins with the group type T_Statements. This type contains a subtype which for its part begins with a specific type T_Var. Both, the T_Statements and the T_Var have to occur only once which is determined by the quantifier. The T_Var is the AST representation of the variable block and contains a number of variable initialisations. Each of them appears in the AST as a general type T_Assign. The type T_Statements is the body of the variable block and contains a number of any statements. Additionally to this pattern, a technique has to be introduced to define the convention constraint. This technique needs to check if each variable which appears within the program also appears in the head of the variable block.

3. **AST Types**

The third constraint addresses the convention which regards the avoidance of AST types within a program. This constraint uses a technique which is based on a depth-first-search to check if there are some of these prohibited types in the AST. If the initial program P_0 of a program transformation process does not contain one of the unwanted AST types, it is also possible to prohibit the applications of FermaT

transformations which are capable to create them. The identification of these transformations with the aid of transformation capabilities will be discussed in detail in Chapter 5.

Apart from structural conventions, there are a lot of other conventions which can be inspected via constraints. For instance, very common and well known code conventions are naming conventions which define that module names and classes should be nouns and should begin with an upper-case letter whereas procedures, functions and methods should begin with a verb and a lower-case letter. Currently, these conventions are only achievable with human interaction because the maintainer has to define a suitable name for the procedures, functions and methods within a program. There exist also no FermaT transformation which is able to rename modules at all. Furthermore, classes can only be defined by using some of the latest extensions of WSL [60].

However, the wide spectrum type system which has been published by Matthias Ladkau solves the variable declaration problem [51]. As discussed in Chapter 3 there exist only a prototype tool of this type system which has a lot of limitations.

4.3 High-Level (Structural) Constraints

High-level (structural) constraints are addressing global characteristics or a set of characteristics of the source code. As mentioned before, they are checked by inspection. In contrast to low-level (structural) constraints, it is often possible to use approved measuring techniques like complexity or structural metrics to define high-level (structural) constraints. Furthermore, a lot of other constraints are based on them which includes some environmental (structural) constraints.

4.3.1 Abstraction Level Constraints

WSL has been developed to provide a wide abstraction spectrum within a single language which has been discussed in Chapter 3. Therefore, it is possible to write low-level language statements as well as high-level language statements and even specification statements within one and the same WSL program. On the one hand, this is required if individual FermaT transformations are used to change the abstraction level of a program as is often the case with the migration of legacy systems [74]. On the other hand, it can lead to

problems if there are still low-level statements after the program transformation has been finished. These problems affect not only the comprehensibility of the source code. They can also cause problems during the translation to other languages. This is the main reason to restrict the abstraction level of a program by abstraction level constraints.

Apart from ensuring the comprehensibility and the translatability of a WSL source code, there are other program transformation tasks which benefit from abstraction level constraints. For example, these constraints can also be very important to satisfy other constraints. A good example is the LoC reduction of a program after it has been translated from an Assembler language. To achieve this, it is often necessary to raise the abstraction level of the source code by transforming an included action system which contains low-level language constructs into high-level language constructs. Afterwards, some more FermaT transformations can be applied which eminently benefit from the previous program transformation and massively reduce the LoC. However, the raise of the abstraction usually causes a huge increase of the LoC. Therefore, it is difficult to find suitable transformation sequences. This applies in particular if it is done automatically via search tactics. In other words, it is difficult to implement search tactics which take interplay effects into consideration.

The problem gets even worse if the interplay effects are widespread and appear in the scope of a few hundred or thousand applied FermaT transformations [31]. In this case, the maintainer could assist the particular search tactic by introducing an abstraction level constraint which first raises the abstraction level and than decreases the LoC. This is called a milestone strategy where a constraint has to be satisfied to guide the satisfaction of another constraint. The inclusion of constraints into a program transformation process will be discussed in Chapter 6.

An approach to introduce abstraction level constraints is to separate the AST types of the WSL language into several different abstraction level groups. Afterwards, constraints can restrict the abstraction level of a WSL program through prevention of AST types which belong to other than the selected abstraction level group. This technique requires a classification of each involved AST type. It is related to convention constraints which prohibit some of these types within the source code of a WSL program.

At the current stage, the separation only concerns the specific types of the WSL language. These have been split into three groups. The first group contains low-level specific types whereas the second group contains high-level specific types. The last group contains specific types which cannot be assigned to a particular abstraction level. These types can be considered as abstraction level independent. The constraint in this case would be either *"low-level language"* which means there must not be any AST types in the WSL program which belong to the group of high-level AST types or *"high-level language"* which means there must not be any AST types in the WSL program which belong to the group of low-level AST types. The detailed assignment of the particular specific types into one of these groups depends a lot on the characteristics of the source code and can slightly vary case-by-case. Therefore, the passage below discusses an example which compares the AST types of a low-level WSL program with the AST types of a high-level WSL program. The example starts with the following listing of a low-level WSL program.

Listing 4.8: Low-Level WSL Program Example.

```
1  ACTIONS PROG:
2    PROG ==
3      C:"Action PROG";
4      CALL A
5    END
6    A ==
7      C:"Action A";
8      IF i < j THEN
9        i := i + 1;
10       k := k + i;
11       CALL PROG
12     FI;
13     CALL B
14   END
15   B ==
16     C:"Action B";
17     IF i > 10 THEN
18       CALL Z
19     FI;
```

```
20      CALL A
21    END
22 ENDACTIONS
```

The low-level WSL program consists of an action system which contains three actions. There is one comment and at least one call within each action. This example illustrates how WSL code looks after the translation from an Assembler language and after some basic restructuring. There are already *IF* statements within the program but loops are implemented via calls. Furthermore, there is an action system which represents the structure of the original Assembler program which has been discussed in Chapter 3.

The advantage of this program is certainly that a maintainer which is already familiar with the Assembler code will recognise the translated WSL program and will easier get used to it. On the other hand, high-level language is more abstract and generally easier to comprehend. The following listing shows another WSL example program. This is equivalent to the program which has been shown in the previous listing in terms of denotational semantics. It has been generated automatically via a program transformation which has been carried out with the aid of the FTE.

Listing 4.9: High-Level WSL Program Example.

```
1 C:"Action PROG";
2 DO
3   C:"Action A";
4   IF i < j THEN
5     i := i + 1;
6     k := k + i
7   ELSIF i > 10 THEN
8     C:"Action B";
9     EXIT(1)
10  FI
11 OD
```

The high-level WSL program consists of a *DO* loop which contains an *IF* statement. The action system as well as the calls and the actions themselves have disappeared. The comments are still present but they became inconsistent during the program transformation process. In general, the source code of the program is shorter and matches the com-

mon expectations on high-level code in terms of the abstraction from the Assembler language. The following table compares the specific types of the low-level and the high-level WSL programs.

Table 4.2: Comparison of the Low-Level and High-Level Specific Types within the WSL Program Examples.

Specific Type	**Number (Low-Level Program)**	**Number (High-Level Program)**
T_Assignment	2	2
T_A_S	1	0
T_Call	5	0
T_Comment	3	3
T_Cond	2	1
T_Exit	0	1
T_Floop	0	1
T_Greater	1	1
T_Less	1	1
T_Name	4	0
T_Number	2	2
T_Plus	2	2
T_Variable	6	6
T_Var_Lvalue	2	2

As the table shows, most of the specific types appear in both versions of the WSL program. These can be assigned to the third defined group which contains abstraction level independent types. The specific type T_A_S appears only in the low-level code. This type certainly belongs to the low-level group. In contrast, the type T_Floop appears only in the high-level code. Therefore, this type belongs as well as every other loop type to the high-level group. The specific type T_Exit only appears in correlation with *DO* loops. For this reason, it can also be considered as high-level type. The T_Call is one of these types which cannot always be classified. In this particular case, it belongs to the group of low-level types but there are cases where calls can appear in high-level language as well. As mentioned before, a detailed classification of the particular specific types into groups is not always unambiguously and depends much on the characteristics of the

source code of a program.

4.3.2 Analysis and Comprehension Constraints

In the area of software evolution, it is often necessary to analyse and comprehend the source code of a program. For example, this is a typical problem during the migration of a legacy system. A program is often so extensive and complex that a single maintainer is simply not able to understand every little detail. As a matter of fact, this is usually not necessary. In general, the responsible maintainer automatically generates abstract perspectives of a program with the aim to understand its structure. Once this has been accomplished, the maintainer uses a kind of top-down approach to identify the critical statements within this program. Afterwards, these statements can be analysed and comprehended [18].

The idea behind analysis and comprehension constraints is to support a maintainer during this investigation process. If the source code of a program has a high complexity or its structure is difficult to understand, it is not always helpful to change the perspective how the program is looked at. It might be better to keep the perspective and change the program via semantic preserving transformations. This is because a maintainer is generally more interested in the behaviour of a program rather than in its appearance. For example, if a maintainer wants to comprehend an abstract perspective which is the control flow graph of a program, the constraint can be defined as complexity limit the program must not exceed. This can be measured by the cyclomatic complexity metric which is directly related to the control flow graph. Afterwards, a transformation sequence can be applied to decrease the cyclomatic complexity of the program.

In general, analysis and comprehension constraints are very abstract which makes them hard to define. They also depend a lot on the particular characteristics of a given program. At the current stage, these constraints are used in combination with complexity metrics. The purpose of these metrics is to provide a possibility to measure the complexity of a program in terms of a particular perspective. For a proper use of these constraints, it is beneficial to define such a complexity metric for each abstract perspective which is used during program analysis and comprehension processes. Moreover, it is conceivable to use other techniques like for example program slicing in combination with these constraints

to hide unimportant information from the maintainer [98].

4.3.3 Data Constraints

Program transformation is often used to assist a maintainer with the challenges of software evolution. This assistance is widespread and covers several different tasks. However, the WSL language was developed with the ulterior motive to be a flexible language and an independent basis for program transformation processes. Despite that, it is used almost exclusively for the migration of legacy systems. A reason for this unilateral use is the absence of a type system in WSL. This absence causes that data types have to be transferred separately from the source language to the target language during a migration project which is an extensive task. Moreover, the translation from WSL to a target language which supports data types is a lot more complicated if there is no source language at all [49].

A proper solution to solve this problem is the introduction of a type system for WSL. As mentioned before, this has been published by Mathias Ladkau in 2009 [51]. The proposed approach of this work is a wide spectrum type system which introduces data types via typing rules. These rules can also be considered as convention constraints. Furthermore, the proposed introduction of so-called WSL type layers can be considered as data constraints. It might also be possible to use constraints to introduce access modifier or to describe the adaption of data layouts within WSL programs. The endianness is a famous example for this application area.

Data constraints are classified as high-level constraints because they consist of a set of sub constraints which propagate through the source code. For instance, it is almost impossible to introduce types at a single point of the code and leave the rest unchanged. Usually, type safety is consistent through the entire code and most languages provide only one class of type system. However, there currently exist no strict definition of data constraints. Moreover, there is only a very small number of FermaT transformations which are able to satisfy these constraints within the FTE. Nevertheless, data constraints might become very important in combination with future extensions of the WSL language.

4.3.4 Metric Constraints

In general, it is impossible to control program characteristics or properties which cannot be measured [23]. Therefore, metrics have become an important facility to control the quality of software. A software metric can be defined as a function which describes a particular program characteristic or property by means of a numeric value [24]. A metric constraint in turn can be considered as a mathematical interval which has to include this value to be satisfied. To put it briefly, metric constraints use software metrics to define the measurable requirements on a program.

As mentioned before, a software metric addresses a particular program characteristic or property. Since high-level (structural) constraints are defined to be used only in combination with characteristics, there are also metric constraint which belong to the group of behavioural constraints. For example, one of these metrics is the actual complexity metric. This counts the independent paths which are traversed during the execution of a program.

The following example shows a metric constraint which uses the Cyclomatic Complexity Metric (CCM) m. This metric is defined as the number of edges e minus the number of nodes n plus two times the number of the connected components p of a control flow graph [63]. It expresses the complexity of the decision structure of a program as numeric value. In this example, the constraint is satisfied if this number is greater than 15 and less than 50 where this interval has been determined on the basis of personal experience with the comprehension of WSL code:

$$m = e - n + 2 * p, 15 < m < 50$$

Cyclomatic complexity is not the only metric which can be used in correlation with constraints. As a matter of fact, there are already several different metrics in use which include the structural metric as well as the module design complexity metric. The structural metric gives a weighted sum of the structural features of a source code whereas the module design complexity metric reflects the complexity of calling patterns of a module. However, each metric has been implemented directly in $\mathcal{META}$WSL and is therefore part of the FTE. Furthermore, new metrics can be defined as needed. For example, the count

of a specific program characteristic which can simply be the number of a particular AST type or the LoC can be considered as a metric.

Metrics can also be appropriate for the definition of other constraints. For instance, it is imaginable to define a metric to measure how qualified the source code of a program is in terms of a particular compiler or a particular programming language. This is important in regard to the environmental constraints which will be discussed in the following section.

4.4 Environmental (Structural) Constraints

As mentioned before, the FTE allows a maintainer to adapt software by mathematically proven transformations which can be applied on a WSL program [83]. This ensures that the denotational semantics of a program are preserved even after its appearance has been changed. On the other hand, there are structural characteristics which are predefined by the environment and which are not affecting the denotational semantics. Therefore, environmental (structural) constraints are defined as constraints which are determined by the environmental influences on the structure of a program.

4.4.1 Compiler Constraints

In general, a compiler is an application which translates a program from a source language into a target language. Furthermore, the source language is often a high-level language which is abstracted from the machine whereas the target language is usually a low-level language which is machine-oriented [3].

Nowadays, there are a lot of different compilers available for most of the common languages. For example, three of the major compilers for the C language are the GNU C Compiler (GCC), the Intel C Compiler (ICC) and the Microsoft C Compiler (MCC). In these cases, the C example is particularly suitable because C is one of the languages which can be generated from WSL via a automated translation [99].

Due to the variety of different hardware and operating systems, there exist various releases of one and the same compiler. Furthermore, a compiler is software as well and is

therefore continuously evolving [57]. For this reason, there are often a number of versions of a particular compiler for the same hardware and operating system available. The result is that the constraints on the source code which has to be compiled can change not only across different compilers. These constraints can also change in terms of different releases of the same compiler for different platforms and even with regard to different versions of a particular compiler which have been developed for one and the same platform. The following listing shows an example which illustrates this issue with the aid of a program which contains critical C code.

Listing 4.10: Compiler Critical Cast-as-Lvalue Example in C.

```
1 char* p;
2 ((int*) p)++;
```

The presented *cast-as-lvalue* example has been deprecated as of GCC v3.4 and is prohibited as of GCC v4.0 because it was hard to comprehend [75]. Therefore, a source code which is using this kind of typecast has to be changed or cannot be compiled with the latest versions of the GCC. Another listing shows the *conditional-expression-as-lvalue* example which has been deprecated as of GCC v3.4 and is prohibited as of GCC v4.0 as well.

Listing 4.11: Compiler Critical Conditional-Expression-as-Lvalue Example in C.

```
1 int a, b, c;
2 (a ? b : c) = 2;
```

This example has been removed from the GCC because it was considered as C unlike. There exist a lot more compiler extensions and restrictions which have been enforced during the evolution of the application. The dissimilarities between different compilers are even worse. Moreover, it is sometimes possible to compile a source code on different compilers but the resulting programs behave differently because of data type mismatches and so on.

Therefore, the task of compiler constraints is to describe the restrictions of different compilers and compiler versions for different platforms. They can be used to adapt a source code for the requirements and the known weaknesses of each particular compiler. Therefore, this class of environment constraints consist of a set of low-level constraints which define the prohibited code characteristics. However, there currently exist no imple-

mentation of compiler constraints for WSL.

As mentioned before, the problems during a compiling process are often caused by data types. Furthermore, FermaT transformations can only be applied on WSL code. Afterwards, it is possible to automatically translate the WSL code into a target language. This leads to the fact that many compiler problems which are caused by data types cannot be solved on the basis of the current version of the FTE. Once more, the reason is that the original approach of WSL does not support a type system. In fact, this problem does not occur by using the prototype tool of the wide spectrum type system for WSL despite all the limitations of this tool [51].

However, it is questionable if other compiler related problems like the *cast-as-lvalue* example or the *conditional-expression-as-lvalue* example can be solved by compiler constraints. This is because these problems are often very language dependent and cannot be modelled in WSL that easy.

4.4.2 Programming Language Constraints

Programming Language Constraints are regarding the syntax and the semantic of a programming language. They are related to compiler constraints although they rely on a language specification like the C International Standard Specification [43] instead of a compiler implementation. Furthermore, programming language constraints are facing similar problems as compiler constraints.

Because of the use of WSL as intermediate language, the critical syntax is actually generated during the translation to the target language. Moreover, an important part of the semantic is covered by type systems which are introduced during the translation process in terms of the current version of the FTE [51]. Nevertheless, statements which have to be avoided in the target language can be traced back to specific WSL statements. Therefore, it is often possible to transform the WSL source code of a program to avoid the generation of critical statements within the target language during the translation process. To put it briefly, the intention of programming language constraints is to define a set of characteristics which is a subset of the WSL language characteristics and can be translated to a selected language.

4.5 Low-Level (Behavioural) Constraints

In modern times, the maintenance and the correctness of high-level source code has usually a much higher priority than the efficiency of the compiled program in terms of execution speed and memory consumption. Once the program has been finished, the compiler is supposed to optimise the source code and to generate highly efficient Assembler code. Moreover, the optimisations which will be carried out during the compilation process can be controlled by so-called optimisation options [77, 43]. For example, the ICC which is widely known for its abilities to generate adaptable Assembler code has the following general optimisation options [42].

Table 4.3: General Optimisation Options of the ICC v11.

Option	**Optimisation**	**Comment**
/Od, -O0	None	This option is the default setting
/O1, -O1	Code Size	This option creates the smallest code in most cases
/O2, -O2	Execution Speed	This option creates fast code
/O3, -O3	Aggressive Execution Speed	This option creates the fastest code in most cases. It uses the optimisation of the previous option plus some more aggressive loop and memory-access optimisations. These include techniques like scalar replacement, loop unrolling, code replication to eliminate branches, loop blocking to allow a more efficient use of the cache and additional data prefetching
/Zi, -g	Debug	This option generates optimised or unoptimised code with debug information for use with any of the common platform debuggers

In addition to the presented general options, there are a lot of other optimisations which adapt the generated Assembler code to a particular environment or for a particular

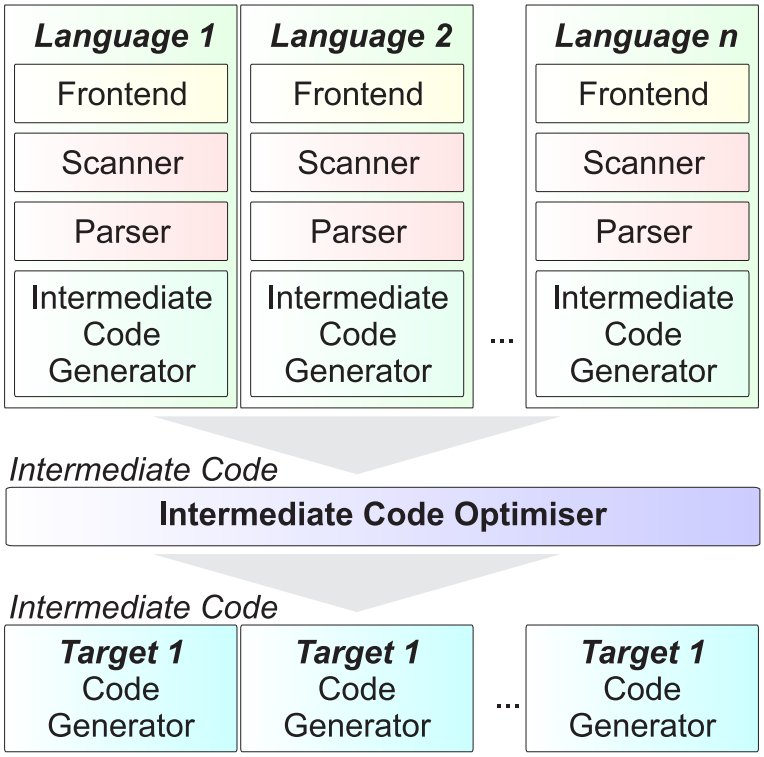
purpose. As a result, the compiler becomes more and more a code optimiser which is actually not its original intention. Therefore, the idea is to use WSL and the extensive set of FermaT transformations for behavioural optimisations which are defined by constraints. Once a program has been optimised, it can be compiled without extensive optimisations which leads to much simpler compilers. Furthermore, this idea could be used to enhance the optimisation capabilities of a compiler which will be discussed in Chapter 9.

The separation of code optimisation and compilation has another advantage. The optimisation part can be done in greater extend because it is not as time critical as the compilation part and it is not needed during the development stage. Moreover, the actual program compilation process is a lot faster because it can be reduced to the basic essentials. Moreover, compilers for new languages or for new platforms can be implemented more easily because the optimisation part is completely independent from it.

However, it has to be mentioned that most of the current compilers are so-called multi language - multi target compilers. These are already using an intermediate language which uncouples the optimisation from the compilation part. Nevertheless, the optimisation process is usually not as profound as in the presented approach of this thesis. Furthermore, the multi language - multi target compiler concept is the reason why modern compilers are often not as fast as they could be even if all optimisations are deactivated [36]. Figure 4.2 shows the architecture of such a compiler.

In terms of the FTE, low-level (behavioural) constraints are a key component to optimise a particular behaviour of a program. Their intention is to provide a possibility to take influence on various atomic, local program properties. In general, low-level (behavioural) constraints are used in combination with high-level (behavioural) constraints. This is necessary because a program alteration can only be achieved by the application of FermaT transformations on the given program. Due to the fact that these transformations are only transforming WSL code, low-level (behavioural) constraints are in a way the connection between the source code and the behaviour of a program. More precisely, these constraints attempt to define structural changes of the program which are beneficial for a particular behaviour whereas high-level (behavioural) constraints are used to control these structural changes in an abstract manner. For example, metric constraints can be used to determine the maximum number of execution speed constraints which are allowed

Figure 4.2: Architecture of a Multi Language - Multi Target Compiler.



to be unsatisfied within a particular program. This will be discussed in Chapter 9 which presents i.a. the use of low-level and high-level (behavioural) constraints to increase the execution speed of a program.

As a matter of fact, each low-level (behavioural) constraint has a corresponding structural constraint. This is because low-level (behavioural) constraints as well as structural constraints address a specific structural element of a program. The difference is that low-level (behavioural) constraints have the purpose to improve a particular program behaviour whereas structural constraints have the purpose to improve a particular program structure. However, structural constraints are based on characteristics. Therefore, the relation of low-level (behavioural) constraints to these constraints leads to a major problem. This major problem is caused by the fact that the satisfaction of structural constraints is checked by inspection. An inspection in turn is impossible for some properties because they depend on an unknown environment. For instance, it is possible that an execution speed constraint is satisfied on some systems but not on others.

For this reason, the proposed approach of behavioural constraints which depend on such an unknown environment is assumption-based and uses low-level (behavioural) constraints which attempt to address the desired behaviour. In other words, the intention of low-level (behavioural) constraints is that the maintainer uses these constraints to define the structural changes which are assumed to lead to his program transformation targets in terms of behavioural alterations. This approach has the advantage that the satisfaction of a low-level (behavioural) constraint can also be checked by inspection because it is nothing more than the definition of a structural program alteration. On the other hand, the definition of such a constraint itself must assure that the structural changes really provide the desired behavioural improvement which can be a very difficult task.

4.5.1 Execution Speed Constraints

In terms of execution speed constraints, it is assumed that program properties which take a certain amount of time to execute are slowing down the execution speed of the entire program. Accordingly, it is also assumed that a reduction of the number of these properties causes an increase of the execution speed. Therefore, the purpose of execution speed constraints is to define individual properties which take a certain amount of execution time.

Once this definition has been finished, a program transformation process can be used to eliminate the defined properties. The expected result is an improvement of the program execution speed although it cannot be guaranteed that the satisfaction of execution speed constraints will lead to the desired effects on every machine. This is because of three reasons which can be summarised as follows:

1. An elimination of program properties which are addressed by the given execution speed constraints may introduce other program properties which are not addressed by the given execution speed constraints. These newly introduced properties can have a negative influence on the program execution speed.

2. There are too many environmental dependencies which are crucial for the execution speed. These dependencies include not only the individual architecture of the processor, the number of processors, the cache hierarchy, the memory size and the

operating system but also the used compiler or the virtual machine on which the program is supposed to run. The problem of environmental dependencies becomes even worse if the environment is unknown.

3. The program properties which are addressed by the given execution speed constraints play only a minor role or are not executed at all. For example, the elimination of a simple assignment outside a loop which runs several hours will have almost no effect.

As mentioned before, the purpose of execution speed constraints is to define individual properties which take a certain amount of execution time. For example, these properties can be assignments, calls, conditions, loops or specific mathematical operations. In general, execution speed constraints address only a fraction of a program. Therefore, they are classified as low-level constraints. The effect of some FermaT transformation on the execution speed of a program will be discussed in Chapter 5. This effect is crucial for execution speed constraints.

4.5.2 Memory Consumption Constraints

Memory consumption constraints are closely related to execution speed constraints. In contrast to them, the purpose of these constraints is to define individual properties which are responsible for an increased memory consumption of a program. Unfortunately, the similarities between these constraint classes mean that they are facing similar problems. As well as for execution constraints, it cannot be guaranteed that the satisfaction of these constraints will lead to the desired effects. Furthermore, the memory consumption of a program is usually not constant during its application.

Additionally to the problems which are mentioned above, memory consumption constraints are related to data constraints and depend a lot on the used data types. This fact causes additional problems because the original approach of WSL does not support a type system at all [49]. For that reason, the introduction of constraints which correlate with data types is nearly impossible. Therefore, these constraints currently address only the amount of variables and the size of the program itself. However, the wide spectrum type system which has been published by Matthias Ladkau solves this problem [51].

4.6 High-Level (Behavioural) Constraints

High-level (behavioural) constraints are addressing global properties or a set of properties of a program. As mentioned before, a constraint of this class is satisfied if the addressed property is an element of the set of program properties or if the set of addressed properties is a subset of the set of program properties. There are also some high-level (behavioural) constraints which can be checked by inspection because they are based on low-level (behavioural) constraints.

4.6.1 Metric Constraints

A software metric can be defined as a function which describes a particular program characteristic or property by means of a numeric value [24]. A metric constraint in turn can be considered as a mathematical interval which has to include this value to be satisfied. Since high-level (behavioural) constraints are defined to be used only in combination with properties, there are also metric constraint which belong to the group of structural constraints. These have been discussed in one of the previous sections.

However, properties are generally harder to measure than characteristics. Furthermore, they depend very much on the environment. Therefore, metrics in terms of behavioural constraints are often based on the satisfaction of low-level (behavioural) constraints. More precisely, metric constraints are used to measure the structural changes which have been carried out by FermaT transformations where each particular structural change has been defined on the basis of a low-level (behavioural) constraints. For example, metric constraints can be used to determine the maximum number of memory consumption constraints which are allowed to be unsatisfied within a particular program.

As mentioned before, the intention of low-level (behavioural) constraints is that the maintainer uses these constraints to define the structural changes which are assumed to lead to his program transformation targets in terms of behavioural alterations. The advantage of this approach is that the satisfaction of low-level (behavioural) constraints can be checked by inspection. Since some of the metric constraints are based on these low-level (behavioural) constraints, they can also be checked by inspection. This relationship in turn leads to some disadvantages which can be summarised as follows:

1. It is difficult to define metric constraints which address a particular behaviour.

2. It is almost impossible to prove that the satisfaction of a metric constraint which uses the defined low-level constraints leads to the desired behaviour due to the environmental dependencies.

However, the Chapter 9 discusses the successful attempt to increase the execution speed of a program with the aid of low-level and high-level (behavioural) constraints.

Apart from the use in combination with low-level (behavioural) constraints, there is another application field for metric constraints which belongs to the class of behavioural constraints. In contrast to structure oriented metric constraints, they are regarding the application of a program. Therefore, they can be used in combination with runtime metrics such as the actual complexity metric. This metric counts the independent paths which are traversed during a program execution. Due to the fact that such metrics are based on the semantics of a program, they can be considered as independent from the environment. Therefore, it is possible to check the satisfaction of a corresponding constraint by inspection as well.

4.6.2 Runtime Constraints

Runtime constraints are based on a technique which directly measures the addressed program property. For example, if a runtime constraint is used to increase the execution speed of a program, this technique has to measure the execution speed by any means. The advantage of this approach is that a satisfaction of this constraint is proven to lead to the desired effect on the addressed program behaviour. The disadvantage is that the satisfaction can only be regarded in relation to the satisfaction context which includes the given environment.

However, it is difficult to implement runtime constraints because the corresponding measuring technique is working during the execution of a program. Due to the proposed approach which is based on WSL, the program has to be translated all the way down to executable code. Afterwards, it is necessary to execute the program while the addressed property is measured. Furthermore, the measuring technique itself could distorts the measuring results.

4.7 Environmental (Behavioural) Constraints

A software never runs independent from its environment and there are always restrictions which cannot be ignored. Furthermore, hardware and software has become very complex during the last decades and a program has to be adapted at regular intervals to keep up with its environment. This has been discussed in Chapter 2. As mentioned before, the FTE provides various ways to adapt software by mathematically proven transformations which can be applied on a WSL program [83]. This ensures that the denotational semantics of a program is preserved even after some properties of it have changed. On the other hand, there are behavioural properties which are predefined by the environment and which are not affecting the denotational semantics. Therefore, environmental (behavioural) constraints are defined as constraints which are determined by the hardware environment or the software environment of a program.

To take up the compiler example of the low-level (behavioural) constraint again, a compiler is often able to adapt the generated code for a specific processor type, a specific processor architecture or a specific instruction set. It is also possible to fine-tune the code via a few dozen specific optimisations for different kinds of code and environment. The question here is if it makes sense to optimise a program via the compiler after it has actually been finished or is it better to start the optimisation during the software evolution process, where the maintainer can directly influence the code?

4.7.1 Hardware Constraints

The hardware environment has a strong influence on several different properties. Furthermore, if the hardware environment of a system is known, it is often much easier to define structural changes which are beneficial for a particular property. Therefore, there are two main approaches of hardware constraints in the scope of this thesis.

It is often predictable which kind of structure changes improve an addressed property. Therefore, the first approach is based on low-level (behavioural) constraints and for this reason closely related to the approach of metric constraints. More precisely, a particular hardware constraint can be considered as metric constraint which is adapted to measure the fitness of a program in terms of a specific hardware environment. In contrast, the second approach is related to runtime constraints. These constraints are particularly suit-

able because the hardware environment defines the hardware constraints and is therefore known. Nevertheless, runtime constraints are depending on the entire environment and are therefore even more suitable if hardware and software constraints are used in combination.

Hardware constraints are covering the entire hardware environment in which a program is supposed to run. This includes the machine itself which usually consist of one or more processor cores as well as a specific amount of memory and a couple of peripheral devices. Also, it includes the system architecture which can be helpful to reduce the influences of bottlenecks during the program execution. For example, it is often possible to adapt the size of a critical loop to the internal cache structure of a processor [38].

4.7.2 Software Constraints

Software constraints are closely related to hardware constraints. These constraints are using the same two main approaches but adapted for the software environment. In general, they can be considered as the complement of hardware constraints and are therefore often used in combination. However, software constraints are covering the entire software environment in which a program is supposed to run. This includes the operating system and the underlying or collaborating programs.

4.8 Summary

The presented chapter has defined program transformation constraints. It has introduced a classification of these constraints which splits them into structural and behavioural constraints. For each of these classes, the differences between low-level, high-level and environmental constraints have been explained. Moreover, it has discussed the importance and the benefit of these constraints for a successful use of program transformation theory. Program transformation processes have been observed from different perspectives and the relation of constraints to program characteristics and properties has been discussed. Furthermore, the dependence of constraints to various measuring techniques such as pattern matching and metrics have been presented and explained.

Chapter 5

Capabilities and Effects of Transformations

Objectives

- To introduce transformation capabilities and effects.
- To describe their extraction and to discuss their dependencies.
- To reveal their value for practical applications.

The FermaT Transformation Engine (FTE) provides a set of mathematical proven WSL to WSL transformations. These so-called FermaT transformations themselves are programs written in $\mathcal{META}$WSL. In general, these programs consists of two procedures which are in particular an applicability condition and a transformation definition. The applicability condition determines the structure of the WSL code on which the corresponding FermaT transformation can be applied. In general, it can be considered as pattern matching definition based on essential and complex $\mathcal{META}$WSL commands. The transformation definition in turn describes how the corresponding FermaT transformation modifies the structure of a program. It is also based on $\mathcal{META}$WSL commands but generally a lot more complex than the applicability condition. Transformation theory and

$\mathcal{M}_{ETA}$WSL have been described in Chapter 3 whereas an overview of the FermaT transformations will be provided in Appendix B.

However, this chapter introduces transformation capabilities and effects. These are useful for a lot of different tasks which includes the comprehension of FermaT transformations as well as the modelling of program transformation processes. On the one hand, transformation capabilities are very complex and difficult to extract. They are static and have to be captured directly from the $\mathcal{M}_{ETA}$WSL code of the respective transformation. To simplify this procedure, transformation capabilities will be considered as five different sets of AST types in the scope of this thesis. On the other hand, transformation effects are a lot easier to extract but they are dynamic and depend on the program which has to be transformed as well as the constraints which have to be satisfied. More precisely, they are defined as differences of some specific program characteristics or properties before and after a particular FermaT transformation has been applied. The affected characteristics or properties in turn are determined by a dedicated constraint. To simplify the use of transformation effects, they will be considered as ratings in the scope of this thesis. These ratings define the respective effect in terms of a given constraint. They can be extracted through a comparison of the initial and the transformed source code of a program.

Transformation capabilities and effects are closely related and depend on each other. In fact, it is often the case that an effect is directly caused by a particular capability of the FermaT transformation which is applied on a given program. However, there are also some transformation effects which do not have a corresponding transformation capability. This is often the case with effects in relation to program properties. For example, the effect of a FermaT transformation on the execution speed of a program which is running on a particular system can be measured with the aid of both program versions P_i and P_{i+1} but it cannot be extracted from the source code of the respective transformation. On the other hand, the effects of a FermaT transformation applied on a particular program reflect only a subset of the capabilities of the transformation. This is because effects depend a lot on the given program.

At the current stage of the presented research, only a special set of transformation capabilities and effects are of interest. As mentioned before, these can be helpful to model program transformation processes which will be described in Chapter 6. Moreover, trans-

formation capabilities are particularly meaningful for a prediction technique which will be discussed in Chapter 7. Transformation effects in turn are important for the evaluation of transformation sequences which will be discussed in Chapter 7 as well.

5.1 Definition of Transformation Capabilities

As mentioned before, transformation capabilities are extracted directly from the $\mathcal{META}$WSL code of a FermaT transformation. Accordingly, they are not depending on the given program which has to be transformed and they are not depending on any constraints which have to be satisfied either. For this reason, they can be considered as unalterable and therefore as static.

However, transformation capabilities are defined as sets of AST types in the scope of this thesis. This approach is very simple and can be implemented easily. It is also very fast to check if a particular AST type is in one of these sets which is very practical. At the current stage of the research, there exist five sets which are defined as follows:

1. The first set A describes the capability on which AST types the FermaT transformation can be applied. Accordingly, the set A contains all AST types on which the transformation is possibly applicable. If the applicability condition checks for a more complex program structure, only the selected AST type is considered. For example, the Delete All Skips transformation which has been discussed in Chapter 3 is applicable on any AST type. Additionally, it checks if the selected AST type which indicates the root of a defined AST contains the specific type T_Skip. However, this check for a more complex program structure will not be taken into account which means that there are all AST types of WSL in the set A.

2. The second set C_C describes the capability of a FermaT transformation to create AST types for certain. Accordingly, the set C_C contains AST types which will certainly be created after a successful application of a transformation. Furthermore, it contains the number of each certainly created type. For example, the set C_C of the While to Floop transformation contains one time the specific type T_Floop which will be created in any case but it does not contain the specific type T_And which creation depends on the given program.

3. The third set C_P describes the capability of a FermaT transformation to possibly create AST types. Accordingly, the set C_P contains AST types which will possibly be created after a successful application of a transformation. For example, the set R_P of the While to Floop transformation contains the specific type T_And which creation depends on the given program but it does not contain the specific type T_Floop which will be created in any case.

4. The fourth set R_C describes the capability of a FermaT transformation to remove AST types for certain. Accordingly, the set R_C contains AST types which will certainly be removed after a successful application of a transformation. Furthermore, it contains the number of each certainly removed type. For example, the set R_C of the While to Floop transformation contains one time the specific type T_While which will be removed in any case but it does not contain the specific type T_Or which removal depends on the given program.

5. The fifth set R_P describes the capability of a FermaT transformation to possibly remove AST types. Accordingly, the set R_P contains AST types which will possibly be removed after a successful application of a transformation. For example, the set R_P of the While to Floop transformation contains the specific type T_Or which removal depends on the given program but it does not contain the specific type T_While which will be removed in any case.

The sets C_C and R_C also contain the number of AST types which will be created or removed. However, a FermaT transformation can only be applied on a selected AST type of a particular program if the applicability condition returns true. The application of a FermaT transformation in turn has been successful if it has not been aborted by the *ERROR* command. An unsuccessful application of a transformation is usually caused by a WSL syntax error.

5.2 Relation of Constraints and Transformation Effects

As mentioned before, transformation capabilities can be considered as unalterable and therefore as static because they are extracted directly from the $\mathcal{META}$WSL code of a FermaT transformation. They are not depending on the given program which has to be transformed and they are not depending on any constraints which have to be satisfied

either. In contrast, transformation effects are defined as program alterations which are caused by the application of a FermaT transformation. These alterations can be captured by a comparison of the program state P_i and another program state P_{i+1} which has been generated by a particular transformation. Moreover, a transformation effect is defined as change in regard to a given constraint.

Constraints are always used in combination with measuring techniques. As a matter of fact, the entire classification of constraints and their overall practical value depends a lot on them. On the one hand, a measuring technique returns a measuring result to express a particular characteristic or property of a given program. On the other hand, a constraint defines whether a particular measuring result leads to satisfaction or not. For example, a program can be measured by the Cyclomatic Complexity Metric (CCM) which returns a natural number as measuring result. Afterwards, a constraint can be used to define that a measuring result of greater than 15 and less than 50 leads to satisfaction where the particular interval has to be chosen by the maintainer.

The presented approach uses measuring techniques which are relatively simple. In every case, their result is either a boolean or a numerical value. Even if the measuring technique is an object which has not been developed for measuring purposes, the result can be considered as boolean or numerical value. A good example is a compiler which counts the number of errors and warnings during the compilation process. This leads to one or more numerical values where each is addressed by an individual constraint. Another example is a particular hardware or software environment which determines if a program can be executed or not. This leads to a boolean.

The connection of constraints and transformation effects is the described measuring technique. Therefore, the maintainer has to define a positive or a negative effect of a FermaT transformation on the basis of the used measuring technique of a constraint. For example, a positive effect of a FermaT transformation in terms of a CCM based constraint could be a decrease of the cyclomatic complexity whereas a negative effect could be an increase of the cyclomatic complexity of a given program. The same applies for other measuring techniques. For instance, a positive effect of a FermaT transformation in terms of a pattern constraint could be that a particular pattern within a program does not match before but does match after an application of the transformation. On the other hand, a

negative effect could be that a particular pattern within a program does match before but does not match after an application of the transformation.

5.3 Definition of Transformation Effects

As mentioned before, transformation effects are extracted through a comparison of the initial and transformed source code of a program. Accordingly, they are depending on the given program which has to be transformed and they are depending on a particular constraint which has to be satisfied as well. For this reason, they can be considered as alterable and therefore as dynamic.

However, transformation effects will be considered as ratings in the scope of this thesis. Similar to the approach of transformation capabilities, this approach is very simple and can be implemented easily. It is also very fast to check how the effect of an individual FermaT transformation is rated in combination with a particular constraint. At the current stage of the research, there exist seven different rates which can be described as follows:

1. C_j:**P**: *Positive Effect* - The application of the rated FermaT transformation has always a positive effect in relation to the particular constraint C_j. For example, an application of the Delete All Skips transformation deletes at least one *SKIP* statement within the given program. If there is no *SKIP* statement at all, the transformation is inapplicable. If a positive effect is defined as decrease and a negative effect is defined as increase of the Number of Code Characters (NoCC), the effect of the rated transformation on NoCC based constraints is positive.

2. C_j:**PX**: *Positive or No Effect* - The application of the rated FermaT transformation has a positive or no effect in relation to the particular constraint C_j. For example, an application of the Constant Propagation transformation will possibly remove some *IF* statements, *WHILE* loops or other statements from the given program. This in turn would decrease the cyclomatic complexity of the given program. If a positive effect is defined as decrease and a negative effect is defined as increase of the Cyclomatic Complexity Metric (CCM), the effect of the rated transformation on CCM based constraints is either positive or there is no effect at all.

3. C_j:**PN**: *Positive or Negative Effect* - The application of the rated FermaT transformation has either a positive or a negative effect in relation to the particular constraint

C_j. For example, an application of the Rename Local Variables transformation always deletes a *VAR* statement within the given program. If there is no *VAR* statement at all, the transformation is inapplicable. The WSL syntax of a *VAR* statement in turn consists of twelve characters which will be discussed in Appendix A. On the other hand, it extends each variable within the deleted *VAR* statement by seven additional characters which will be discussed in Appendix B. If a positive effect is defined as decrease and a negative effect is defined as increase of the NoCC, the effect of the rated transformation on NoCC based constraints is positive if there is only one variable within the *VAR* statement which does not appear elsewhere in the given program. Otherwise, the effect is negative.

4. C_j:**PNX**: *Positive, Negative or No Effect* - The application of the rated FermaT transformation has either a positive, a negative or no effect in relation to the particular constraint C_j. For example, an application of the Rename Procedure transformation replaces the old name of a procedure within the given program by a new one. If a positive effect is defined as decrease and a negative effect is defined as increase of the NoCC, the effect of the rated transformation on NoCC based constraints is positive if the new name consists of less characters than the old name. It is negative if the new name consists of more characters than the old name. There is no effect at all on NoCC based constraints if the new name consists of the same number of characters than the old name.

5. C_j:**X**: *No Effect* - The application of the rated FermaT transformation has always no effect in relation to the particular constraint C_j. For example, an application of the Insert Assertions transformation inserts one or more assertions into the given program if some suitable information can be ascertained. If there are no such information, the transformation is inapplicable. The introduction of assertions in turn has no effect on CCM based constraints.

6. C_j:**NX**: *Negative or No Effect* - The application of the rated FermaT transformation has a negative or no effect in relation to the particular constraint C_j. For example, an application of the Loop Doubling transformation will duplicate the body of a loop within the given program. This in turn could increase the cyclomatic complexity of the given program. If a positive effect is defined as decrease and a negative effect is defined as increase of the CCM, the effect of the rated transformation on CCM based constraints is either negative or there is no effect at all.

7. C_j:**N**: *Negative Effect* - The application of the rated FermaT transformation has always a negative effect in relation to the particular constraint C_j. For example, an application of the Unroll Loop transformation inserts at least one *IF* statement into the given program which increases the cyclomatic complexity. If a positive effect is defined as decrease and a negative effect is defined as increase of the CCM, the effect of the rated transformation on CCM based constraints is negative.

The definition of a positive and a negative effect in terms of a particular constraint is not trivial. As mentioned before, these effects need to be determined in relation to the corresponding measuring technique of the constraint. In general, the definition is a lot easier for high-level and environmental constraints which are using complex measuring algorithms as for low-level constraints.

5.4 Extraction of Transformation Capabilities

As discussed in Chapter 3, the source code of a FermaT transformation is split into two $\mathcal{META}$WSL procedures. The first procedure is the applicability condition which checks if the transformation is applicable whereas the second procedure is the transformation definition which describes how the FermaT transformation changes the program. Transformation capabilities are directly extracted from these two procedures. Moreover, they are defined as sets which contain AST types in the scope of this thesis. Each of these sets addresses a particular capability. The following listing shows the source code of the While to Floop transformation to demonstrate the extraction of transformation capabilities.

Listing 5.1: $\mathcal{META}$WSL code of the While to Floop transformation to demonstrate the extraction of Transformation Capabilities.

```
1 MW_PROC @While_to_Floop_Test() ==
2   IF @TYPE(@Item) = T_While THEN
3     @Pass
4   ELSE
5     @Fail("test: ast type mismatch")
6   FI
7 END;
8 MW_PROC @While_to_Floop_Code() ==
9   VAR <B := < >, S := < >>:
```

```
10      IFMATCH Statement WHILE ~?B DO ~*S OD THEN
11        B := @Not(B);
12        @Paste_Over(
13          FILL
14            Statement DO IF ~?B THEN EXIT(1) FI; ~*S OD
15          ENDFILL
16        )
17      ELSE
18        ERROR("code: ifmatch error")
19      ENDMATCH
20    ENDVAR
21  END
```

The source code is split into the *While_to_Floop_Test*() procedure which is the applicability condition and the *While_to_Floop_Code*() procedure which is the transformation definition. In addition to standard WSL statements, the applicability condition uses the essential commands of $\mathcal{META}$WSL to check if the selected AST type is the specific type T_While. If this is the case, it uses the *@Pass* command to terminate the procedure and to return true. Otherwise, it uses the *@Fail*(*m*) command to terminate the procedure and to return false and an error message. The important detail for the presented approach of transformation capabilities is the *IF* condition. Only if the selected type is a T_While, the FermaT transformation is applicable. Therefore, the set *A* only contains one AST type which is the mentioned T_While. In other words, the first capability of the While to Floop transformation is that it can only be applied on the specific type T_While. The transformation definition in turn uses i.a. the essential and the complex commands of $\mathcal{META}$WSL to convert a *WHILE* loop into a *DO* loop. The four sets R_C, R_P, C_C and C_P are extracted from this procedure. The *FILL* statement within the $\mathcal{META}$WSL code indicates that the *WHILE* loop certainly will be removed after the application of the transformation. Therefore, the set R_C contains the specific type T_While. Moreover, the *WHILE* loop contains a head and a body. These are only shifted and will not be removed while the FermaT transformation is executed. For this reason, the set R_C only contains one type.

More precisely, the *WHILE* loop will not be removed but overwritten by a *DO* loop which in turn contains other AST types. Moreover, the condition which is placed in the

head of the *WHILE* loop will be changed as appropriate and inserted into the *DO* loop as condition of an *IF* statement. In contrast, the body of the *WHILE* loop will be taken over unchanged. For this reason, the set C_C contains the general type T_Guarded, the group type T_Statements and the specific types T_Cond, T_Exit and T_Floop.

At first sight, it seems that all capabilities which are of interest for this approach have been extracted. However, there is a little detail which must not be unappreciated. The condition *B* of the loop will be negated during the transformation process. This is done by the $\mathcal{META}$WSL command @Not(B) which constructs the condition $\neg B$ and then tries to simplify it. This command is able to remove as well as to create various AST types depending on the source code which will be transformed. Therefore, each of the sets R_P and C_P contain the specific types T_And, T_Equal, T_Even, T_False, T_Greater, T_Greater_Eq, T_In, T_Less, T_Less_Eq, T_Not, T_Not_Eq, T_Not_In, T_Odd, T_Or and T_True. The number of these types is not indicated within the two sets.

The While to Floop transformation is relatively simple compared to some others. In general, FermaT transformations are more extensive which makes the extraction of capabilities more difficult. In this particular case, the applicability condition only checks the selected type which makes it possible to assure that the While to Floop transformation is applicable if the selected type is in the set *A*. As discussed in Chapter 3, this is not always the case. Therefore, it can often not be assured that a FermaT transformation is applicable if the selected type is in the set *A*. To prove that a transformation is applicable, the applicability condition has always to be executed. On the other hand, it can be assured that a FermaT transformation is not applicable if the selected type is not in the set *A*. However, it has to be mentioned that the capability approach cannot replace the applicability condition. In fact, it is not even the intention of the capability approach to replace this condition.

The advantage of capabilities over effects is that they are directly extracted from the source code of a FermaT transformation which guarantees that all capabilities are certainly there and do not depend on the program which has to be transformed. The disadvantage of capabilities is that they are difficult to extract. This applies particularly in regard to automated extraction. To use capabilities in practical applications, the five sets of AST types have to be created for each of the given FermaT transformations. At the current stage of the research, this has been done for the transformations which have been used in

Chapter 9. To create these sets was required to use the prediction technique which will be discussed in Chapter 7.

5.5 Extraction of Transformation Effects

Transformation effects can be considered as the counterpart of transformation capabilities. On the one hand, capabilities are used to define the operating range of a FermaT transformation. On the other hand, effects are used to measure the impact of such a transformation applied on a particular program.

In contrast to capabilities, effects are defined as differences of program characteristics and properties before and after a particular FermaT transformation has been applied. Therefore, they can be extracted through a comparison of two subsequent program states P_i and P_{i+1} where the second program state P_{i+1} has been created by a FermaT transformation applied on the first program state P_i. This comparison applies in regard to a particular constraint and to the measuring technique the constraint uses.

Transformation effects are dynamic because they depend on three critical factors which can be summarised as follows:

1. The first factor is the constraint which has to be satisfied. As mentioned before, the particular effect of an individual FermaT transformation is defined on the basis of the used measuring technique of a constraint. Therefore, the transformation effect varies depending on the given constraint.

2. The second factor is the program which has to be transformed. The particular effect of an individual FermaT transformation applied on a given program possibly varies depending on the structure of this program.

3. The third factor is the program environment. Some constraints address a particular behaviour which can only be measured in combination with a hardware and software environment. A change of this environment can also cause an alteration of the transformation effect.

The result of these factors is that the application of a particular FermaT transformation can cause different transformation effects which are often unpredictable. The following

listing shows a simple WSL example program on which a FermaT transformation will be applied to demonstrate the extraction of transformation effects.

Listing 5.2: WSL Example Program to Demonstrate the Extraction of Transformation Effects.

```
1 WHILE i = 5 OR j < 10 DO
2   i := i + 1;
3   j := j - i
4 OD
```

The FermaT transformation which will be applied on the WSL example program is the While to Floop. As the name implies, it converts a *WHILE* loop into a *DO* loop. The capabilities of this transformation have been discussed in the previous section. However, the selected AST type within the given program has to be the specific type T_While because it is the only one on which the While to Floop transformation is applicable. The head of the *WHILE* loop which will be converted is a condition *B*. This condition consists of two subconditions B_1 and B_2 which are combined by an *OR* operator. The body of the loop contains two assignments. The following listing shows a program which is the result of the While to Floop transformation applied on the program which has been shown in the previous listing.

Listing 5.3: Transformed WSL Example Program to Demonstrate the Extraction of Transformation Effects.

```
1 DO
2   IF i <> 5 AND j >= 10 THEN
3     EXIT(1)
4   FI;
5   i := i + 1;
6   j := j - i
7 OD
```

As the transformation capabilities define, the *WHILE* loop and along with it the specific type T_While has been removed. Moreover, several types have been created which include the general type T_Guarded, the group type T_Statements and the specific types T_Cond, T_Floop and T_Exit. The body of the loop has been unchanged whereas the condition has been negated. This negation has caused that the specific types T_Equal,

T_Or and T_Less have been removed whereas the specific types T_Not_Eq, T_And and T_Greater_Eq have been created.

The example has shown that transformation effects depend a lot on the transformed program. Also, it has shown how close transformation capabilities and transformation effects are related. However, it is often a lot easier to extract effects than to extract capabilities because the extraction as such is actually a comparison between measuring results of two different programs in relation to a particular constraint.

5.6 Structural and Behavioural Effects

Each FermaT transformation can be considered as a definition of a structural program alteration. The capabilities of a such a transformation in turn are directly extracted from the METAWSL code. For this reason, these capabilities regard the structure of a program and not its behaviour without any exception. As a matter of fact, there is no direct connection between the capabilities of a FermaT transformation and the program which will be transformed during an application of this transformation.

On the other hand, structural changes have often impacts on a particular program behaviour which has been discussed in Chapter 4. These impacts can be measured by a comparison of the initial and the transformed program. Therefore, there exist structural effects as well as behavioural effects of a FermaT transformation. The following listing shows the WSL code of a program before a particular transformation has been applied.

Listing 5.4: WSL Code of an Initial Example Program to Demonstrate a Characteristic Change.

```
1 DO
2   IF i >= j THEN
3     EXIT(1)
4   FI;
5   k := k + i;
6   i := i + 1
7 OD
```

The example consists of a *DO* loop which contains one condition and two assignments. The loop will be exited if the condition is true. The program has an individual structure which can be described as a set of particular characteristics. Furthermore, its application causes an individual behaviour which can be described as a set of properties. Some of these characteristics and properties can be changed by the application of a particular FermaT transformation on the program. The following listing shows a program which is the result of the Loop Doubling transformation applied on the program which has been shown in the previous listing.

Listing 5.5: WSL Code of a Transformed Example Program to Demonstrate a Characteristic Change.

```
1  DO
2     IF  i >= j THEN
3        EXIT(1)
4     FI;
5     k := k + i;
6     i := i + 1;
7     IF  i >= j THEN
8        EXIT(1)
9     FI;
10    k := k + i;
11    i := i + 1
12 OD
```

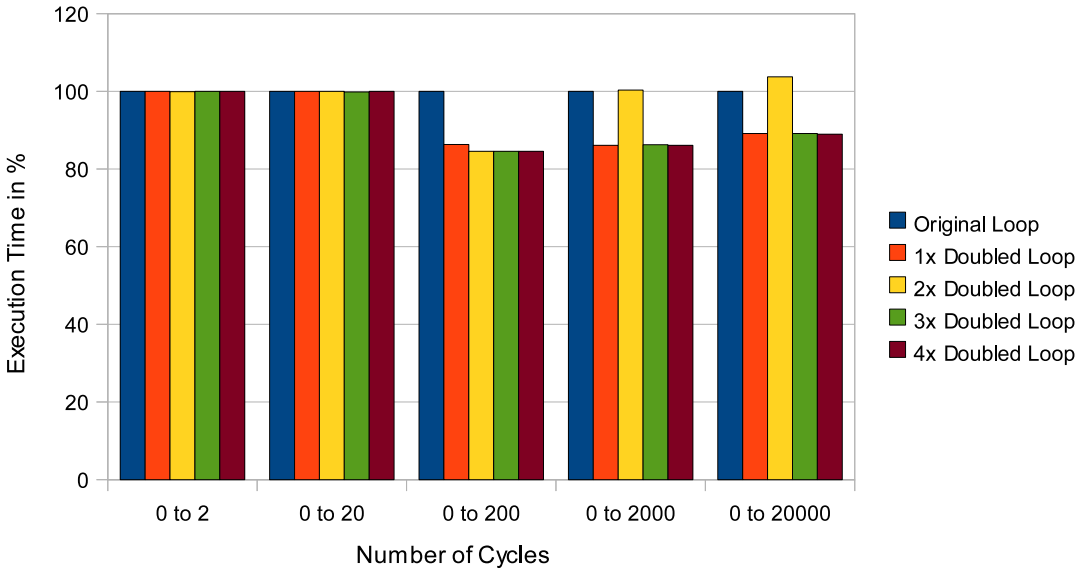
As the listing shows, the body of the *DO* loop has been duplicated. The structure of the program and along with it the program characteristics to describe the structure have been changed. For example, the LoC have been increased from seven to twelve and the cyclomatic complexity of the program has been increased from two to three. Furthermore, it can be assumed that some program properties have been changed as well. The following listing shows the C code of an initial program to demonstrate a particular property change on the basis of the Loop Doubling transformation. In this case, C has been chosen as programming language because WSL programs can only be interpreted and not be compiled. This limits the appropriateness of WSL programs in terms of time-critical comparisons which will be discussed in Chapter 9.

Listing 5.6: C Code of an Initial Example Program to Demonstrate a Property Change.

```
1 for (i = 0; i <= n; i++) {
2   sum += i;
3 }
```

The example consists of a *for* loop which contains one assignment. It calculates the sum of the values 0 ... *n* where *i*, *n* and *sum* are integer variables. Figure 5.1 shows how the execution time of the program has been changed after one, two, three and four applications of the Loop Doubling transformation. To apply this FermaT transformation, the program has to be translated from C to WSL. Afterwards, the resulting program has to be translated back to C.

Figure 5.1: Execution Speed of a Transformed Example Program in Relation to the Initial Program.

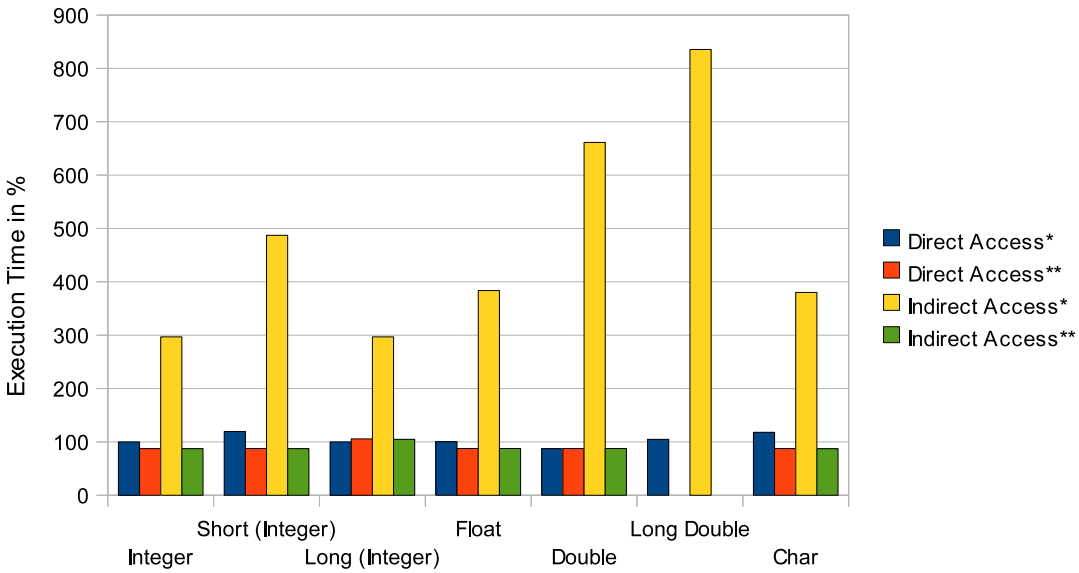


The execution time measurements have been taken on a PC which was equipped with an Intel Core 2 Duo processor, 2 GiB of main memory and Windows XP Professional SP2. The programs have been compiled with the GNU C Compiler (GCC) v4.4.0 where no further optimisation from the compiler has been carried out. In relation to the execution speed constraint and within the given environment, the Loop Doubling transformation is rated as C_3:**PNX** which means its application can cause a positive, a negative or no effect. At least in this case, the average improvement in terms of execution speed which can be

achieved by this FermaT transformation is certainly positive. However, average improvements are irrelevant for the proposed approach of transformation effects which might be a disadvantage.

The example has shown how unpredictable behavioural effects can be and how dependent they are from the given environment. Nevertheless, the approach of behavioural effects is useful because they kind of propose which FermaT transformations could lead to the satisfaction of a given constraint. Therefore, the proposed approach of transformation effects can be considered as assistance to model a program transformation process. Moreover, these effect can also be integrated into these models with the aid of so-called $\mathcal{META}$Constraints. This will be discussed in Chapter 6. Figure 5.2 shows a diagram which presents another example for the effect of decreased execution time.

Figure 5.2: Execution Speed Comparison of Direct and Indirect Variable Access where a single asterisk (*) indicates a C program and a double asterisk (**) indicates a Java program.



The execution time measurements have also been taken on a PC which was equipped with an Intel Core 2 Duo processor, 2 GiB of main memory and Windows XP Professional SP2. Moreover, the programs have also been compiled with the GNU C Compiler (GCC)

v4.4.0 where no further optimisation from the compiler has been carried out. To get some comparative values, the programs have been translated from WSL to Java as well where the Sun Java Development Kit (JDK) v1.6.0_03 has been used to compile and run the Java source code.

The different bars indicate the execution time of direct and indirect variable access where the C program is indicated by a single asterisk (*) and the Java program is indicated by a double asterisk (**). In all cases, the direct variable access is a lot faster than the indirect variable access which was to be expected. However, indirect variable accesses via so-called getter and setter functions or methods are often used within programs which are written in modern languages. The FTE provides various possibilities to transform direct variable access into indirect variable access and vice versa. For example, this can be achieved with the aid of the Substitute and Delete transformation.

5.7 Summary

The presented chapter has discussed a definition of transformation capabilities and effects and has demonstrated their value for practical applications. It described the extraction of transformation capabilities directly from source code of a FermaT transformation as well as the extraction of transformation effects by the comparison of an initial and a transformed program. Furthermore, the differences between structural and behavioural effects have been explained.

Chapter 6

Modelling Program Transformation Processes

Objectives

- To discuss the modelling of program transformation processes.
- To introduce transformation schemes and $\mathcal{META}$Constraints.
- To present a formal language to control program transformation processes.
- To discuss the construction and conversion of transformation schemes.

As discussed in Chapter 4, the proposed approach of the thesis uses different classes of constraints to define program transformation targets. Subsequently, a particular search tactic attempts to find suitable transformation sequences to satisfy these constraints which will be discussed in Chapter 7. As well as with other search based program transformation approaches, it is very difficult to adapt this search tactic which makes it relatively inflexible [22]. Moreover, the search space Ω contains the exponential number of t^n transformation sequences where t is the number of available FermaT transformations and n is the transformation sequence length [30]. For this reason, the search space is often extremely large which makes exhaustive search infeasible [31]. For instance, there are 20^{50} different transformation sequences in the search space for the relatively small number of

20 available FermaT transformations and a transformation sequence length of 50.

As discussed in Chapter 2, this assumption is still very optimistic. It does not take the places into account where the FermaT transformations can be applied within a program. To consider these places would lead to an even larger search space. Therefore, it is crucial to explore new possibilities to improve the search tactic. One possibility is a prediction technique which is based on transformation capabilities. This technique will be discussed in Chapter 7. Another possibility is the evaluation of transformation sequences which is based on transformation effects. This evaluation will be discussed in Chapter 7 as well.

However, the prediction technique and the evaluation of transformation sequences are supposed to improve the search tactic but they are insufficient to completely solve the problem of the large search space. Therefore, this chapter discusses an approach to model program transformation processes with the aid of so-called transformation schemes. These schemes are based on automata theory in combination with constraints. They provide a possibility to include additional information into the automated process of constraint satisfaction. Accordingly, they solve the mentioned inflexibility problem of other search based program transformation processes.

There exist several tools which help to create transformation schemes. These tools will be introduced and discussed in this chapter as well. One of these tools is a formal language to describe transformation schemes. This language is called Transformation Scheme Description Language (TSDL). Another tool is a construction technique which creates a transformation scheme from the language. However, the additional program transformation knowledge which is included in a particular transformation scheme has to be provided by the maintainer. This knowledge can be collected on the basis of transformation capabilities and effects which have been discussed in Chapter 5. Some good examples how to develop and apply transformation schemes will be presented in Chapter 9.

6.1 Definition of Transformation Schemes

As mentioned before, transformation schemes are based on automata theory and therefore consists of scheme states and scheme transitions. Each scheme state contains a set

of program states during the program transformation process. This is because there are usually several ways to traverse a transformation scheme. The initial scheme state S_0 contains the unchanged program P_0 which the maintainer puts into the system whereas the final scheme states contain the program transformation results. The scheme states are connected by scheme transitions which are defined by so-called transition or transformation functions.

As mentioned before, each scheme state contains a set of program states which have to be evaluated. This evaluation can be handled with the aid of constraints. If there are no constraints at all, the system will return the first applicable transformation sequence which is defined by the transformation scheme. It is also possible to return several program transformation results and let the maintainer choose his favourite manually. The transformation scheme and all details of the program transformation can be shown in a graphical window which will be discussed in Chapter 8. Therefore, the maintainer is able to obtain the entire transformation process and can intervene if necessary.

The transformation schemes which are used in this thesis are based on two particular types of automata. These are the Nondeterministic Finite Automaton which supports ε-Transitions (ε-NFA) and the Deterministic Finite Automaton (DFA). An ε-NFA based transformation scheme is defined as 5-tuple (S, Σ, T, I_S, A) consisting of:

1. A finite set of states (S)
2. A finite set of transformations $(\Sigma \subseteq \Sigma_F)$
3. A transition (or transformation) function $(T : S_i \times (\Sigma \cup \{\varepsilon\} \cup \{\lambda(c)\}) \rightarrow P(S_i))$
4. An initial (or start) state $(I_S \in S)$
5. A set of final states $(A \subseteq S)$

A transition which is not changing the program with the result that $S_i \equiv S_{i+1}$ is indicated by the Greek letter ε whereas a transition which is not changing the program but which usage depends on a particular condition c is indicated by the Greek letter λ. In general, the transition function is defined as function from an individual scheme state S_i to a powerset of this state $P(S_i)$. Furthermore, the proposed approach defines Σ to be a

subset of the FermaT transformations Σ_F.

A valid transformation sequence is defined as follows. Let M be a transformation scheme such that $M = (S, \Sigma, T, I_S, A)$ and T_S be a transformation sequence over the set of transformations Σ. The sequence T_S is in the scheme M if there is a representation of the sequence T_S in the form $t_0, ..., t_{n-1}, t_i \in (\Sigma \cup \{\varepsilon\} \cup \{\lambda(c)\})$ and if a sequence of states $S_0, ..., S_n, S_i \in S$ exists which satisfies the following conditions:

1. $S_0 \in I_S$
2. $S_i \in T(S_{i-1}, t_{i-1}), i = 1, ..., n$
3. $S_n \in A$

The set of all transformation sequences which is defined by the transformation scheme is the search space Ω. As discussed in Chapter 4, a transformation sequence starts in an initial program state P_0 which is the only existing program state at this point. This state can be transformed into another program state P_i by the application of a particular FermaT transformation. The program state which has been created by the last transformation t_{n-1} of a transformation sequence T_{Si} is the final program P_n:

$P_0\ t_0\ P_1\ t_1\ P_2\ t_2\ ...\ t_{n-1}\ P_n$

The advantage of ε-NFA based transformation schemes is that they can easily be constructed from formal language via the Thompson Construction [1]. This will be discussed in the following sections. However, the disadvantage is that the transition function of these schemes is defined as function from an individual scheme state S_i to a powerset of this state $P(S_i)$. The result is that ε-NFA based transformation schemes are working with a set of current scheme states rather than a single current scheme state.

The complex transition function makes it difficult for a maintainer to comprehend ε-NFA based transformation schemes. Moreover, it causes some difficulties during further processing. Therefore, there are DFA based transformation schemes as well. These schemes are working with a single current state and do not support ε-transitions. A DFA based transformation scheme is defined as 5-tuple (S, Σ, T, I_S, A) consisting of:

1. A finite set of states (S)

2. A finite set of transformations $(\Sigma \subseteq \Sigma_F)$
3. A transition (or transformation) function $(T : S \times (\Sigma \cup \{\lambda(c)\}) \rightarrow S)$
4. An initial (or start) state $(I_S \in S)$
5. A set of final states $(A \subseteq S)$

A transition which is not changing the program with the result that $S_i \equiv S_{i+1}$ and which usage depends on a particular condition c is indicated by the Greek letter λ. In general, the transition function is defined as function from an individual scheme state S_i to the scheme state S_{i+1}. Furthermore, the proposed approach defines Σ to be a subset of the FermaΓ transformations Σ_F.

A valid transformation sequence is defined as follows. Let M be a transformation scheme such that $M = (S, \Sigma, T, I_S, A)$ and T_S be a transformation sequence over the set of transformations Σ. The sequence T_S is in the scheme M if there is a representation of the sequence T_S in the form $t_0, ..., t_{n-1}, t_i \in (\Sigma \cup \{\lambda(c)\})$ and if a sequence of states $S_0, ..., S_n, S_i \in S$ exists which satisfies the following conditions:

1. $S_0 \in I_S$
2. $S_i \in T(S_{i-1}, t_{i-1}), i = 1, ..., n$
3. $S_n \in A$

Although DFA based transformation schemes are often larger than equivalent ε-NFA based transformation schemes, they are normally easier to comprehend. This is caused by their transformation function which is defined as transition between two scheme states.

6.2 Formal Language for the Description of Transformation Schemes

As mentioned before, transformation schemes can be used to model entire program transformation processes. In other words, transformation schemes allow a maintainer to define a set of transformation sequences as search space. Their appearance is similar to automata

which has many advantages. These advantages include the visualisation of transformation schemes as well as their appropriateness for further processing [40, 2]. However, it is difficult to describe a transformation scheme especially for a human. For this reason, a formal language called TSDL has been developed.

TSDL can be considered as interface for a maintainer to describe transformation schemes. The aim of this language is not only to be powerful enough to describe complex transformation schemes but also to be simple enough and therefore useful for maintainers who are not very familiar with program transformation theory. Once a description has been specified, a slightly adapted version of the Thompson Construction can be used to convert this description into an equivalent transformation scheme. To put it briefly, a transformation scheme appears always in a form similar to an automaton whereas its description can be specified with the aid of the formal language TSDL.

In general, TSDL is a formal language based on regular expressions. Therefore, it supports the conventional sequence and alternative operations of expression languages as well as quantifier and grouping. Moreover, it supports the inclusion of conventional constraints and the definition of transformation sets via so-called $\mathcal{META}$Constraints which will be discussed in Section 6.3. The constraints and $\mathcal{META}$Constraints as well as the FermaT transformations appear within TSDL as the name of their definition. For example, a constraint with the name C_1 which defines that the cyclomatic complexity of a program has to be less than 50 appears within TSDL as C_1. The following table describes TSDL in the Backus-Naur-Form.

Table 6.1: TSDL Description in the Backus-Naur-Form.

BNF Type	**Definition**
<*scheme*>	<sequence>
<*sequence*>	<alternative> ("," <alternative>)*
<*alternative*>	<factor> ("|" <factor>)*
<*factor*>	<subscheme> | <transformation> <quantifier>?

Continued on next page

TSDL Description - continued from previous page.

BNF Type	**Definition**
	<constraints>?
<*subscheme*>	"(" <scheme> ")"
<*transformation*>	"<" <meta_constraints> | FERMAT TRANSFORMATION ("@" AST PATH)? ">"
<*quantifier*>	"[" BINARY NUMBER ".." NATURAL NUMBER "]"
<*constraints*>	"{" CONSTRAINT ("," CONSTRAINT)* "}"
<*meta_constraints*>	"{" (META CONSTRAINT ("," META CONSTRAINT)*)? "}"
FERMAT TRANSFORMATION	a FermaT transformation
AST PATH	a path in the abstract syntax tree of a WSL program
BINARY NUMBER	a single-digit binary number (0 or 1)
NATURAL NUMBER	a natural number (1 to n)
CONSTRAINT	a constraint
META CONSTRAINT	a meta constraint
","	a comma to indicate a sequence
"|"	a vertical bar to indicate a alternative
"("	a left bracket

Continued on next page

TSDL Description - continued from previous page.

BNF Type	**Definition**
")"	a right bracket
"<"	a left angle bracket
"@"	a separator between meta constraints or a FermaT transformation and an AST path
">"	a right angle bracket
"["	a left square bracket
".."	a separator between two numbers to indicate a mathematical interval
"]"	a right square bracket
"{"	a left curly bracket
"}"	a right curly bracket

As the Backus-Naur-Form shows, a *scheme* consists of a *sequence* which for its part consists of *alternatives*. Each of these *alternatives* in turn consists of *factors*. The *scheme*, the *sequence*, the *alternative* and the *factor* are all non-terminals. The *alternatives* of a *sequence* are separated by a comma whereas the *factors* of an *alternative* are separated by a vertical bar. A *factor* consists of a non-terminal which is either a *subscheme* or a *transformation* followed by two optional non-terminals which are a *quantifier* and some *constraints*. A *subscheme* is a *scheme* surrounded by brackets. A *transformation* is a non-terminal which consist of either a particular FermaT transformation or some $\mathcal{META}\mathrm{Constraints}$ to define a transformation set plus an optional AST path which defines where the particular transformation or each transformation within the defined set has to be applied. The FermaT transformation or the $\mathcal{META}\mathrm{Constraints}$ and the AST path are divided by a particular separator and are surrounded by angle brackets. All other non-terminals consist only of terminals and signs.

An AST path selects a particular AST type or a set of AST types within a program. In this case, selection means that a specific FermaT transformation or a set of transformations has to be applied on the particular AST type or on the set of AST types. However, it is not required to state the AST path of each particular FermaT transformation and each transformation set within the described transformation scheme. If the path is not stated,

there is simply no restriction which limits the application area of the transformation or the transformation set. The application area in turn is defined as a particular AST type or a set of AST types on which a specific FermaT transformation has to be applied.

In other words, the applicability of a particular FermaT transformation or a transformation set which is defined by $\mathcal{META}\mathrm{Constraints}$ has to be checked on each AST type of the entire program if no AST path has been stated. Furthermore, the transformation or the set of transformations have to be applied on each AST type where the preceding applicability check has been successful. This leads often to a massive consumption of computing power especially in combination with complex transformation schemes and large programs which have to be transformed. For this reason, it is often very profitable to state an AST path.

In general, an actual AST path is surrounded by slashes to identify it within TSDL. In terms of abstract restricted AST Paths, the AST type has to be stated before the actual path. There are five different possibilities to state an AST path:

1. **Definite AST Path** - The AST path is stated as sequence of AST type indices. In the AST of a given program, the corresponding AST types to these indices have to be visited to reach the selected AST type. For example, the definite AST path // is the root type of an AST whereas the definite AST path /0/ is the first subtype of the root type. The definite AST path /0,0/ in turn is the first subtype of the first subtype of the root type whereas the definite AST path /0,1/ is the second subtype of the first subtype of the root type.

2. **Restricted AST Path** - The AST path is stated as a definite AST path where the last index of the sequence is a placeholder. This can be either an asterisk, a plus or a question mark. The asterisk indicates that the AST type at the stated AST path as well as all subtypes are selected whereas the plus indicates that only the subtypes are selected. The question mark indicates that only the direct subtypes of the AST type at the stated AST path are selected. For example, the restricted AST path /0,*/ selects the AST type at the definite AST path /0/ as well as all subtypes of this type whereas the restricted AST path /0,+/ selects only the subtypes of the AST type at the definite AST path /0/. The restricted AST path /0,?/ in turn selects only the direct subtypes of the AST type at the definite AST path /0/.

3. **Unrestricted AST Path** - The AST path is stated as an asterisk or it is not stated at all. If it is not stated at all, the separator which divides a FermaT transformation or a transformation set and the AST path according to the Backus-Naur-Form of TSDL will also not be stated. For example, the non-terminal *transformation* which contains an unrestricted AST path can appear either in the form < ... > or in the form < ... @ /*/ > but not in the form < ... @ >. As mentioned before, the application area of a particular FermaT transformation or a transformation set on an unrestricted AST path includes the entire program which leads often to a massive consumption of computing power.

4. **Abstract AST Path** - The AST path is stated on the basis of an AST type. Therefore, the AST has to be searched to locate this AST type within a given program. Afterwards, a particular FermaT transformation or a transformation set can be applied on each of the locations where the stated AST type has been found. This will be discussed in Chapter 7. For example, the abstract AST path T_While selects all AST types within a given program which are of the kind T_While. However, an abstract AST path will not be surrounded by slashes because it is not a sequence of AST type indices and with it no actual AST path.

5. **Abstract Restricted AST Path** - The AST path is stated on the basis of an AST type on which a FermaT transformation or a transformation set has to be applied. Furthermore, the search to locate this AST type will be performed on a restricted AST path rather than on the entire program as is the case with an abstract AST path. For example, the abstract restricted AST path T_While : /0,+/ selects all AST types within the program which are of the kind T_While and which are a subtype of the AST type at the stated AST path.

At some point in the constraint based program transformation process, a stated AST path has to be decomposed. In other words, restricted as well as unrestricted and abstract AST paths have to be converted into a set of definite AST paths. This will be discussed in Chapter 7. The following listing shows a concrete example of a transformation scheme description written in TSDL.

Listing 6.1: Description Example of a Single Transformation on a given AST Path in TSDL.

```
1 < Floop to While @ /0/ >
```

The example is very simple and shows a transformation scheme which consists of only one particular FermaT transformation. This is the Floop to While which is applied on the AST path /0/. It transforms a *DO* loop into a *WHILE* loop. In this case, the path is stated as a definite AST path. The following listing shows a WSL example program on which the transformation scheme can be applied.

Listing 6.2: Transformation Scheme Application Example in WSL.

```
1 DO
2   IF i >= 10 THEN
3     EXIT(1)
4   FI;
5   i := i + 1
6 OD
```

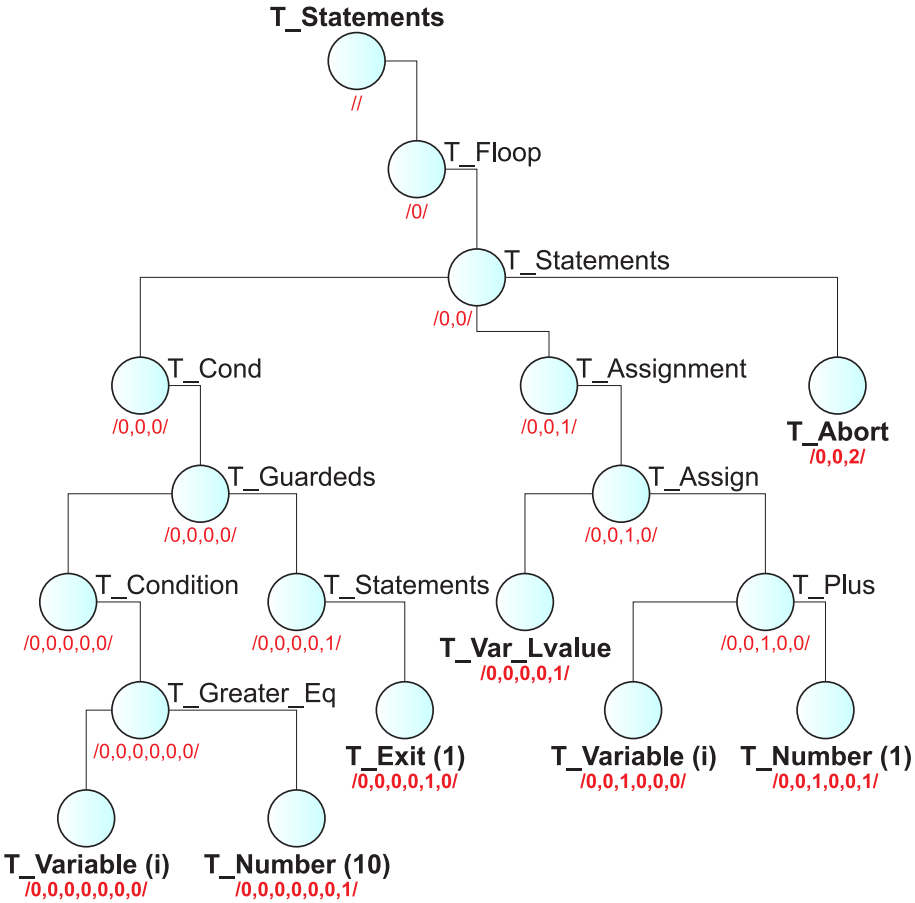
The first AST type and therefore the root of this WSL example program is the group type T_Statements. This is described in Appendix A. The first subtype of the root and with it the AST path /0/ is the specific type T_Floop which is addressed by the Floop to While transformation within the transformation scheme. The first subtype of the T_Floop in turn is another group type T_Statements which therefore has the AST path /0,0/. Figure 6.1 provides an overview about all AST paths of the shown WSL example.

As mentioned before, there are various possibilities to state an AST path. Some of them require an analysis of the AST of a given program. As well as the constraint satisfaction itself, this leads to a search for AST types on which the respective transformation is applicable. Moreover, it often causes a set of different program states as result of the application of an individual transformation sequence. The search for transformation paths as well as the application of transformation sequences will be discussed in Chapter 7.

To take the previous transformation scheme example up again, the defined set of transformation sequences consist of only one FermaT transformation and there is no constraint included at all. Also, there is a static AST path given. Therefore, a search is not necessary. The following listing shows a WSL program which is the result of the Floop to While transformation applied on the defined AST path within the program which has been shown in the previous listing.

Listing 6.3: Result of a Transformation Scheme Application Example on the WSL Pro-

Figure 6.1: AST Path of the WSL Program shown in Listing 6.2.



gram shown in Listing 6.2.

```
1 WHILE i < 10 DO
2     i := i + 1
3 OD
```

The specific type T_Floop has been replaced by another specific type which is the T_While. Furthermore, a lot of program properties and characteristics have been changed whereas others have been unaffected. Because this example is very simple, some of the characteristics and properties of the programs can easily be compared. For example, the LoC has been decreased from six to three. Also, the branching behaviour of the program has been changed which probably has an influence on the execution speed. Nevertheless, there are also some characteristics and properties which remain unchanged. For instance, the cyclomatic complexity of the program is one of them.

However, if a maintainer wants to achieve one of the described program characteristics or properties without any knowledge about the effect of FermaT transformation applications, it is still possible to define a transformation scheme by using TSDL. The following listing shows how such a definition could look like.

Listing 6.4: Description Example of a Transformation Set in TSDL.

```
1 < {} > {C1}
```

The transformation scheme defines that any transformation of the set of FermaT transformations Σ_F can be applied on a given program. This is indicated by the empty set of $\mathcal{META}$Constraints. After the program transformation process, the final program has to satisfy the defined constraint C_1. There is no AST path and no quantifier given. For this reason, each transformation sequence which is defined by this scheme contains only one FermaT transformation. This can be applied on any existing AST path. For example, if the given program is the one which is shown in Listing 6.2 and the constraint C_1 is to decrease the LoC to less than four then a transformation sequence which contains only the Floop to While transformation is in the set of constraint satisfying sequences. In other words, this transformation sequence is valid.

As mentioned before, the use of an unrestricted AST path as is the case in the previous listing leads often to a massive consumption of computing power. Therefore, it is important to state the AST path as precisely as possible. For instance, if the maintainer knows the AST path on which a transformation set has to be applied, it is possible to write a definition similar to the one which is shown in the following listing.

Listing 6.5: Description Example of a Transformation Set on a given AST Path in TSDL.

```
1 < {} @ /0/ > {C1}
```

In this case, the search effort is restricted not only by the limited application area of the transformation set but also by the applicability conditions of the involved FermaT transformations. This is because most of them are probably not applicable on the stated AST path. However, all of the depicted schemes define transformation sequences with a length of only one transformation. Commonly, these schemes will not lead to acceptable results. Therefore, the following listing shows a transformation scheme which defines sequences with the length of two. This is still a relatively small scheme but it shows the idea of a TSDL sequence.

Listing 6.6: Description Example of a Transformation Sequence on given Paths in TSDL.

```
1 < Floop to While @ /0/ >,
2 < While to Floop @ /0/ >
```

The transformation scheme contains a sequence of two FermaT transformations which are separated by a comma. Because there are no cycles and no alternatives, there is only one transformation sequence in the defined search space. This sequence consists of two transformations which are applied only once on the same AST path. Because the While to Floop transformation is a converse of the Floop to While transformation, the application of the defined transformation sequence has no effect. Also, it includes no constraints. The following listing shows a similar transformation scheme which involves the same transformations.

Listing 6.7: Description Example of a Transformation Alternative on given Paths in TSDL.

```
1 (
2   (
3     < Floop to While @ /0/ > |
4     < While to Floop @ /0/ >
5   ) [0 .. 2]
6 ) {C1}
```

The transformation scheme is an alternative of two FermaT transformations which are separated by a vertical bar. In contrast to the example which has been shown in the previous listing, this one includes a quantifier and a constraint. Therefore, the search space for a transformation sequence which satisfies the constraint C_1 contains four transformation sequences. The outer brackets indicate that the transformation scheme has to satisfy the constraint C_1 at the end of the program transformation process and not in the middle of the process.

All FermaT transformations within the defined transformation sequences are applied on the AST path /0/ but only one transformation sequence can be applied on the program which has been shown in Listing 6.2. However, the applicable sequence is the same as in the previous transformation scheme example which has no effect. Therefore, there exist only a valid transformation sequence within the scheme if the constraint C_1 has already been satisfied on the initial program. The following listing shows a transformation scheme

which combines an alternative and a sequence of FermaT transformations.

Listing 6.8: Description Example of a combined Transformation Alternative and Sequence on given Paths in TSDL.

```
1 (
2   < Dijkstra Do to Floop @ /0,0,0/ > |
3   < While to Floop @ /0,0,0/ >
4 ),
5 < Double to Single Loop @ /0/ >
```

The transformation scheme contains an alternative of two FermaT transformations which are the Dijkstra Do to Floop and the While to Floop. Both transformations are applied on the AST path /0,0,0/. Furthermore, the alternative is the first part of a sequence. The second part is a Double to Single Loop transformation applied on the AST path /0/ which is separated from the alternative by a comma. There is no constraint included in this transformation scheme but the last FermaT transformation has to be applied anyway which defines that one double *DO* loop will be transformed into a single *DO* loop if it is possible to apply one of the defined transformation sequences.

However, the presented examples are very small and aim at a specific problem. In practise, a description of a transformation scheme is usually much more extensive. Nevertheless, the examples provide an overview of the possibilities of the language and discusses the key features like FermaT transformations in combinations with AST paths, sequences and alternatives of FermaT transformations as well as quantifier and the inclusion of constraints.

6.3 Definition of Transformation Sets

As mentioned before, a transformation scheme defines a set of transformation sequences. This set represents the search space in which the maintainer supposes a valid transformation sequence. To simplify the development of a transformation scheme, it is possible to use $\mathcal{META}$Constraints to define transformation sets rather than to select a specific FermaT transformation. This has been shown to some extent in Listing 6.4 and in Listing 6.5 without actually using a particular $\mathcal{META}$Constraint.

However, $\mathcal{META}$Constraints are defined on the basis of transformation capabilities

and effects which have been discussed in Chapter 5. A good example to demonstrate the use of such $\mathcal{META}$Constraints is the elimination of an action system within a given WSL program. This elimination can be achieved through a manual selection of a particular FermaT transformation. However, a requirement for this manual selection is that the responsible maintainer knows exactly which transformation has to be applied for the desired effect. The following listing shows an example of a transformation scheme which contains a manual selected FermaT transformation to eliminate an action system.

Listing 6.9: Description of a concrete Transformation on a given Path in TSDL to eliminate an Action System.

```
1 < Collapse Action System @ /0/ >
```

In this case, the Collapse Action System transformation is applied on a particular AST path. As mentioned before, the maintainer has to know exactly which FermaT transformation has to be applied and where it has to be applied. This can be a very difficult task especially in combination with large programs. However, it is possible to define a transformation scheme similar to the following listing.

Listing 6.10: Description of a Set of Transformations on a given AST Path and a Constraint in TSDL to eliminate an Action System.

```
1 < {} @ /0/ > {C1}
```

The transformation scheme defines a transformation set with the aid of an an empty set of $\mathcal{META}$Constraints as it has already been shown in Listing 6.5. Each FermaT transformation within this set can be applied on the AST path /0/. However, the defined transformation set is not restricted and contains any transformation of the set of FermaT transformations Σ_F. Therefore, the search space of this transformation scheme is equivalent to the number of transformations in the set Σ_F. However, this kind of unrestricted transformation sets might be simple to use for a maintainer but they also increase the search effort a lot. The following listing shows a similar example which includes a $\mathcal{META}$Constraint.

Listing 6.11: Description of a Set of Transformations on a given Path via a $\mathcal{META}$Constraint in TSDL to eliminate an Action System.

```
1 < {mC1} @ /0/ > {C1}
```

In contrast to the previous example, the set of $\mathcal{META}$Constraints is not empty. It includes a $\mathcal{META}$Constraint mC_1 to define a transformation set Σ_{mC1}. This set contains

each FermaT transformation t_i of the set Σ which satisfies the given $\mathcal{META}\mathrm{Constraint}$ mC_1:

$$\forall i,\ t_i\ \in \Sigma_{mC1} iff\ t_i\ sat\ mC1,\ t_i\ \in \Sigma$$

As mentioned before, $\mathcal{META}\mathrm{Constraints}$ are defined on the basis of transformation capabilities and effects. The current example regards one of the transformation capabilities. These are defined as five sets in the scope of this thesis where each set contains a number of different AST types which has been discussed in Chapter 5. One of these sets is R_C which contains the AST types that will certainly be removed by an application of the particular FermaT transformation. Therefore, the $\mathcal{META}\mathrm{Constraint}$ mC_1 within the current example is defined to be satisfied if the specific type S_{AST} is in the set of removed AST types R:

$$t_i\ sat\ mC_1\ iff\ S_{AST}\ \in R$$

The specific type S_{AST} would be the T_A_S in this case. To put it briefly, the Listing 6.11 has the same effect as the Listing 6.10 but the transformation set is much smaller due to the included $\mathcal{META}\mathrm{Constraint}$. For instance, this leads to smaller search spaces and therefore to decreased search effort during the subsequent search which will be described in Chapter 7.

As mentioned before, a lot of other capabilities and effects can be used as base to define $\mathcal{META}\mathrm{Constraint}$. This can be very useful especially for maintainer with less knowledge about transformation theory.

6.4 Construction of Transformation Schemes

As mentioned before, the proposed approach of this thesis uses transformation schemes which are based on automata theory. These schemes can be described with the aid of a formal language called TSDL which simplifies the description and therefore the development of such schemes. Once a maintainer has finished a transformation scheme description, an automated construction technique is required which converts the TSDL code into an equivalent transformation scheme.

In the field of automata theory, there exist a lot of different techniques to create an automaton from a formal language [40]. In the presented approach, a slightly adapted version of the Thompson Construction is used. This technique has been developed by Ken Thompson in 1968 and provides a set of constructs for a stepwise creation of automata from regular expressions [79]. The resulting transformation scheme of the adapted Thompson Construction is based on an ε-NFA which has a relatively structured appearance [2]. This can be beneficial in terms of comprehension and visualisation purposes. After the creation process, it is possible to transform the ε-NFA based transformation scheme into a DFA based transformation scheme. This in turn has advantages during further processing [48].

The original Thompson Construction provides five constructs to create an ε-NFA. These are the basic construct, the option construct, the repeat construct, the alternative construct and the sequence construct [1]. The slightly adapted version of this construction technique which is used in the scope of this thesis provides adapted option and repeat constructs which are called quantifier construct type I, type II and type III. The technique also provides an additional construct which is called the subscheme construct. Moreover, it allows the assignment of constraints to a particular scheme state. However, each of the defined constructs has a related non-terminal within the Backus-Naur-Form of TSDL. The implementation of the defined constructs will be discussed in Chapter 8.

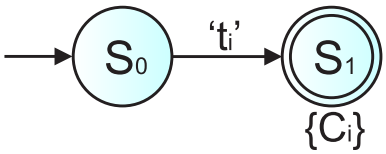
6.4.1 Basic Construct

The basic construct is related to the non-terminal *transformation* which is defined in the Backus-Naur-Form of TSDL. It describes a particular FermaT transformation or a set of transformations as transition between two scheme states where each of these states contains a set of program states. The number of program states within a scheme state depends on the paths how the particular scheme state can be reached.

There is only one program state within the initial scheme state S_0 of the basic construct. This program state is the initial program P_0. Furthermore, the number of program states within the final scheme state S_1 is either one if the transition t_i is a particular FermaT transformation or it is equivalent to the number of FermaT transformations within the de-

fined transformation set if the transition t_i is a set of $\mathcal{META}$Constraints. Figure 6.2 shows an example of a basic construct.

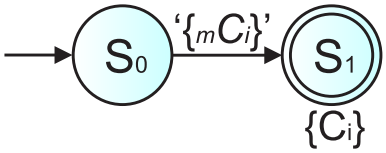
Figure 6.2: FermaT Transformation within a Basic Construct.



This basic construct includes a particular FermaT transformation t_i. The unnamed arrow within this transformation scheme which points on the scheme state S_0 indicates the initial scheme state I_S. This initial scheme state in turn contains the initial program P_0. There is only one final scheme state within this transformation scheme which is indicated by a double circle. This final scheme state contains the set of program states which have to satisfy the constraint C_i.

However, a transformation sequence T_S is accepted by this transformation scheme if it ends in the final state S_1. This is relevant to define the search space and does not determine if the particular transformation sequence is valid. As discussed in Chapter 4, a sequence is only valid if each included transformation $t_0 ... t_n$ is applicable and if each included constraint $C_1 ... C_m$ has been satisfied. Figure 6.3 shows another example of a basic construct.

Figure 6.3: $\mathcal{META}$Constraint within a Basic Construct.



This basic construct includes a transformation set which has been defined on the basis of the $\mathcal{META}$Constraint mC_i. In general, basic constructs are simple transformation schemes which are used as basis for further construction. As a matter of fact, they are the only constructs which are generated completely from scratch. All other constructs are

only extending one or combining two existing transformation schemes. For this reason, there is a marked first state and a marked last state of each subscheme during a construction process. These marked scheme states indicate where the constructed scheme has to be extended or combined. In terms of the five provided constructs of the adapted Thompson Construction, the first marked scheme state is the state S_0 whereas the last marked scheme state is the state with the highest number S_n.

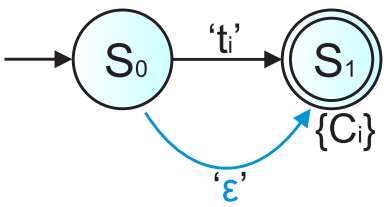
6.4.2 Quantifier Construct

The quantifier construct is related to the non-terminal *quantifier* which is defined in the Backus-Naur-Form of TSDL. It describes a quantification of an existing transformation scheme by adding an ε-transition, a λ-transition or both to this existing scheme. As mentioned before, the three types of this construct can be considered as adapted option and repeat constructs. The type I and the type II of this construct add only one scheme transition to an existing transformation scheme whereas the type III adds two scheme transitions. However, none of them adds any scheme states to an existing transformation scheme. Therefore, there is no need to change the initial and the final state or the first and the last state which are needed for further construction.

Type I

The quantifier construct of type I is similar to the original option construct of the Thompson Construction [79]. It defines that the surrounded transformation scheme can be applied zero to one time. Figure 6.4 shows an example of a quantifier construct of type I. This construct surrounds the basic construct which is shown in Figure 6.2.

Figure 6.4: Basic Construct surrounded by a Quantifier Construct of Type I.



The unnamed arrow within this transformation scheme which points on the scheme state S_0 indicates the initial scheme state I_S. This initial scheme state in turn contains

the initial program P_0. There is only one final scheme state within this transformation scheme which is indicated by a double circle. This final scheme state contains the set of program states which have to satisfy the constraint C_i. Furthermore, the constructed transformation scheme provides two possibilities to reach the final scheme state S_1 from the initial scheme state S_0. The first possibility is via the particular FermaT transformation t_i which is either not applicable or changes the program P_0. The second possibility is via the ε-transition which is defined as not changing the program P_0.

The search space which is defined by the transformation scheme contains two transformation sequences. The first of these sequences contains no FermaT transformation at all. It is valid if the program state P_0 satisfies the constraint C_i. Accordingly, this program state is the initial program P_0 and the final program P_n at the same time. The second of these sequences contains the FermaT transformation t_i. It is valid if the program state P_1 satisfies the constraint C_i. The following listing shows the defined search space of the transformation scheme which has been shown in Figure 6.4. In this listing, the constraint C_i is surrounded by curly brackets and indicates where it has to be satisfied in a particular transformation sequence.

Listing 6.12: Defined Search Space of the Type I Quantifier Construct.

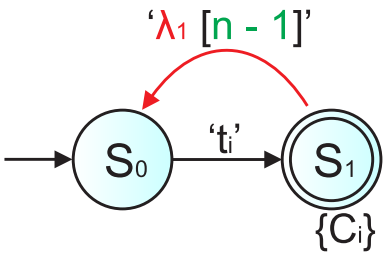
```
1 Sequence 1: {Ci}
2 Sequence 2: ti, {Ci}
```

Type II

The quantifier construct of type II is similar to the original repeat of the Thompson Construction [79]. It defines that the surrounded transformation scheme can be applied one to n times. Figure 6.5 shows an example of a quantifier construct of type II. This construct surrounds the basic construct which is shown in Figure 6.2.

The unnamed arrow within this transformation scheme which points on the scheme state S_0 indicates the initial scheme state I_S. This initial scheme state in turn contains the initial program P_0. There is only one final scheme state within this transformation scheme which is indicated by a double circle. This final scheme state contains the set of program states which have to satisfy the constraint C_i. Furthermore, the constructed transformation scheme provides only one possibility to reach the final scheme state S_1 from

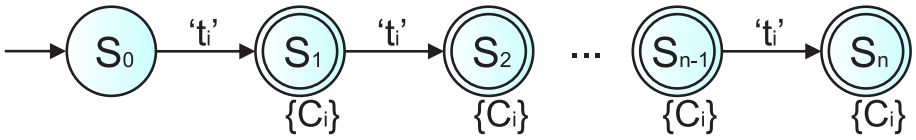
Figure 6.5: Basic Construct surrounded by a Quantifier Construct of Type II.



the initial scheme state S_0. This possibility is via the particular FermaT transformation t_i which is either not applicable or changes the program P_0. On the other hand, it is possible to reach the scheme state S_0 from the scheme state S_1 via the λ-transition $\lambda_1\,[n-1]$ which is defined as not changing the program.

As mentioned before, the usage of this transition depends on a particular condition. In this case, this condition restricts the number how often the λ-transition can be used. More precisely, the particular λ-transition can be used up to $n-1$ times where n is the higher endpoint of the quantifier interval which is defined in the Backus-Naur-Form of TSDL. Therefore, the search space which is defined by the transformation scheme contains n sequences. To clarify this, Figure 6.6 shows how the quantifier construct which is shown in Listing 6.5 looks if it has been unrolled.

Figure 6.6: Alternative View of the Quantifier Construct which is shown in Listing 6.5.



The unnamed arrow within this transformation scheme which points on the scheme state S_0 indicates the initial scheme state I_S. This initial scheme state in turn contains the initial program P_0. Moreover, the unrolled quantifier construct contains $n-1$ final states $S_1 \ldots S_n$ where n has to be greater than one. The final state S_1 can be reached from the initial state S_0 via the particular FermaT transformation t_i whereas the final state S_2 can be reached from the initial state S_1 via the particular FermaT transformation t_i. The final state

S_3 in turn can be reached from the initial state S_2 via the particular FermaT transformation t_i if S_3 exists. In general, the transformation t_i can be used j times to reach the scheme state S_{j+1} from the scheme state S_j where each scheme state S_j contains only one program state which has to satisfy the constraint C_i. The following listing shows the defined search space of the transformation schemes which have been shown in Figure 6.5 and in Figure 6.6. In this listing, the constraint C_i is surrounded by curly brackets and indicates where it has to be satisfied in a particular transformation sequence.

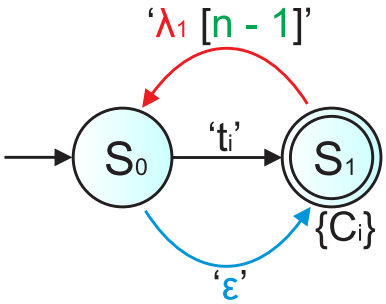
Listing 6.13: Defined Search Space of the Type II Quantifier Constructs.

```
1 Sequence 1: ti , {Ci}
2 Sequence 2: ti , {Ci}, ti , {Ci}
3 Sequence 3: ti , {Ci}, ti , {Ci}, ti , {Ci}
4 ...
5 Sequence n: ti , {Ci}, ti , {Ci}, ti , {Ci}, ... , ti , {Ci}
```

Type III

The quantifier construct of type III is a combination of the quantifier construct of type I and the quantifier construct of type II. It defines that the surrounded transformation scheme can be applied zero to n times. Figure 6.7 shows an example of a quantifier construct of type III. This construct surrounds the basic construct which is shown in Figure 6.2.

Figure 6.7: Basic Construct surrounded by a Quantifier Construct of Type III.

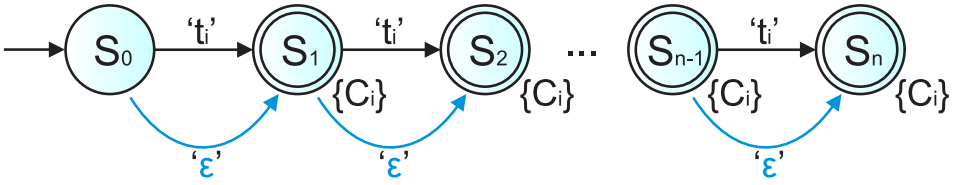


The unnamed arrow within this transformation scheme which points on the scheme state S_0 indicates the initial scheme state I_S. This initial scheme state in turn contains

the initial program P_0. There is only one final scheme state within this transformation scheme which is indicated by a double circle. This final scheme state contains the set of program states which have to satisfy the constraint C_i. Furthermore, the constructed transformation scheme provides two possibilities to reach the final scheme state S_1 from the initial scheme state S_0. The first possibility is via the particular FermaT transformation t_i which is either not applicable or changes the program P_0. The second possibility is via the ε-transition which is defined as not changing the program P_0. On the other hand, it is possible to reach the scheme state S_0 from the scheme state S_1 via the λ-transition $\lambda_1[n-1]$ which is defined as not changing the program.

As mentioned before, the usage of this transition depends on a particular condition. In this case, this condition restricts the number how often the λ-transition can be used. More precisely, the particular λ-transition can be used up to $n-1$ times where n is the higher endpoint of the quantifier interval which is defined in the Backus-Naur-Form of TSDL. Therefore, the search space which is defined by the transformation scheme contains 2_n sequences. To clarify this, Figure 6.8 shows how the quantifier construct which is shown in Listing 6.7 looks if it has been unrolled.

Figure 6.8: Alternative View of the Quantifier Construct which is shown in Listing 6.7.



The unnamed arrow within this transformation scheme which points on the scheme state S_0 indicates the initial scheme state I_S. This initial scheme state in turn contains the initial program P_0. Moreover, the unrolled quantifier construct contains n final states $S_1 \ldots S_n$ where n has to be greater than one. The final state S_1 can be reached from the initial state S_0 via the ε-transition or via the particular FermaT transformation t_i whereas the final state S_2 can be reached from the initial state S_1 only via the particular FermaT transformation t_i. The final state S_3 in turn can be reached from the initial state S_2 via the particular FermaT transformation t_i if S_3 exists. In general, the transformation t_i can be used up to n times to reach the scheme state S_{j+1} from the scheme state S_j depending on

how many ε-transitions have been used. Accordingly, each scheme state S_j contains 2^j program states which have to satisfy the constraint C_i. The following listing shows the defined search space of the transformation schemes which have been shown in Figure 6.7 and in Figure 6.8. In this listing, the constraint C_i is surrounded by curly brackets and indicates where it has to be satisfied in a particular transformation sequence.

Listing 6.14: Defined Search Space of the Type III Quantifier Constructs.

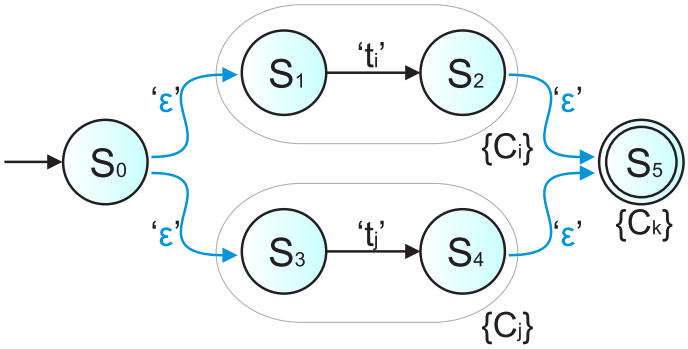
```
1 Sequence 1: {Ci}
2 Sequence 2: ti, {Ci}
3 Sequence 3: {Ci}, {Ci}
4 Sequence 4: {Ci}, ti, {Ci}
5 Sequence 5: ti, {Ci}, {Ci}
6 Sequence 6: ti, {Ci}, ti, {Ci}
7 ...
8 Sequence n: ti, {Ci}, ti, {Ci}, ti, {Ci}, ..., ti, {Ci}
```

6.4.3 Alternative Construct

The alternative construct is related to the non-terminal *alternative* which is defined in the Backus-Naur-Form of TSDL. It describes an alternative between two existing transformation schemes by adding four ε-transitions and two scheme states to these existing schemes. Each of the scheme states within the alternative construct contains a set of program states where the number of program states within a particular set depends on the paths how the particular scheme state can be reached.

There is only one program state within the initial scheme state S_0 of the alternative construct. This program state is the initial program P_0. Furthermore, the number of program states within the final scheme state of the alternative construct is equivalent to the number of program states within the final scheme state of the first existing transformation scheme plus the number of program states within the final scheme state of the second existing transformation scheme. The number of program states within the previously existing scheme states remains unchanged. Figure 6.9 shows an example of an alternative of two basic constructs.

Figure 6.9: Two Basic Constructs combined by an Alternative Construct.



This alternative construct includes a particular FermaT transformation t_i and another particular FermaT transformation t_j. The unnamed arrow within this transformation scheme which points on the scheme state S_0 indicates the initial scheme state I_S. This initial scheme state in turn contains the initial program P_0. There is only one final scheme state within this transformation scheme which is indicated by a double circle. This final scheme state contains the set of program states which have to satisfy the constraint C_k. The initial scheme states S_1 and S_3 and the final scheme states S_2 and S_4 of the existing transformation schemes have been converted into common scheme states. Furthermore, there is a constraint C_i assigned to the scheme state S_2 and there is another constraint C_j assigned to the scheme state S_4. The following listing shows the defined search space of the transformation scheme which has been shown in Figure 6.9. In this listing, the constraints C_i, C_j and C_k are surrounded by curly brackets and indicate where they have to be satisfied in a particular transformation sequence.

Listing 6.15: Defined Search Space of the Alternative Construct.

```
1 Sequence 1: ti , {Ci}, {Ck}
2 Sequence 2: tj , {Cj}, {Ck}
```

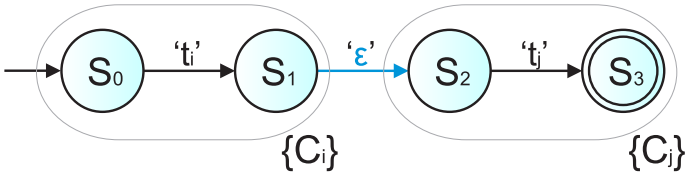
6.4.4 Sequence Construct

The sequence construct is related to the non-terminal *sequence* which is defined in the Backus-Naur-Form of TSDL. It describes a sequence of two existing transformation schemes by adding an ε-transition to these existing schemes. Each of the scheme states within the sequence construct contains a set of program states where the number of pro-

gram states within a particular set depends on the paths how the particular scheme state can be reached.

The number of program states within the initial scheme state S_0 of the sequence construct is equivalent to the number of program states within the initial scheme state of the first existing transformation scheme. Furthermore, the number of program states within the final scheme state of the sequence construct is equivalent to the number of program states within the final scheme state of the first existing transformation scheme times the number of program states within the final scheme state of the second existing transformation scheme. Figure 6.10 shows an example of a sequence of two basic constructs.

Figure 6.10: Two Basic Constructs combined by a Sequence Construct.



This sequence construct includes a particular FermaT transformation t_i and another particular FermaT transformation t_j. The unnamed arrow within this transformation scheme which points on the scheme state S_0 indicates the initial scheme state I_S. This initial scheme state in turn contains the initial program P_0. There is only one final scheme state within this transformation scheme which is indicated by a double circle. This final scheme state contains the set of program states which have to satisfy the constraint C_j. The initial scheme state S_2 and the final scheme state S_1 of the existing transformation schemes have been converted into common scheme states. Furthermore, there is a constraint C_i assigned to the scheme state S_1. The following listing shows the defined search space of the transformation scheme which has been shown in Figure 6.10. In this listing, the constraints C_i and C_j are surrounded by curly brackets and indicate where they have to be satisfied in a particular transformation sequence.

Listing 6.16: Defined Search Space of the Sequence Construct.

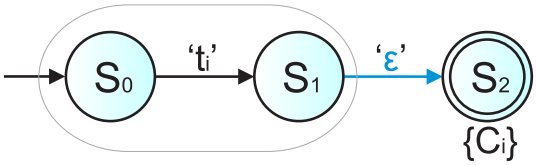
```
1 Sequence 1: ti , {Ci}, tj , {Cj}
```

6.4.5 Subscheme Construct

The subscheme construct is related to the non-terminal *subscheme* which is defined in the Backus-Naur-Form of TSDL. It describes an extension of an existing transformation schemes by adding an ε-transition and a scheme state to this existing scheme. Each of the scheme states within the subscheme construct contains a set of program states where the number of program states within a particular set depends on the paths how the particular scheme state can be reached.

The number of program states within the initial scheme state S_0 of the subscheme construct is equivalent to the number of program states within the initial scheme state of the existing transformation scheme. Furthermore, the number of program states within the final scheme state of the sequence construct is equivalent to the number of program states within the final scheme state of the existing transformation scheme. However, subschemes are necessary to define that a transformation scheme has to satisfy a particular constraint after it has been processed and not while it is processed. Figure 6.10 shows an example of a subscheme which extends a basic constructs.

Figure 6.11: Basic Construct extended by a Subscheme Construct.



This subscheme construct includes a particular FermaT transformation t_i and another particular FermaT transformation t_j. The unnamed arrow within this transformation scheme which points on the scheme state S_0 indicates the initial scheme state I_S. This initial scheme state in turn contains the initial program P_0. There is only one final scheme state within this transformation scheme which is indicated by a double circle. This final scheme state contains the set of program states which have to satisfy the constraint C_i. The final scheme state S_1 of the existing transformation scheme has been converted into a common scheme state.

6.4.6 Transformation Scheme Construction Algorithm

As mentioned before, the slightly adapted Thompson Construction uses the described constructs for a stepwise creation of ε-NFA based transformation schemes from TSDL. The construction algorithm itself is syntax oriented which means that each of the non-terminals *sequence*, *alternative*, *subscheme*, *transformation* and *quantifier* which occurs within a given transformation scheme description causes the execution of the corresponding construction. At each of these so-called construction steps, the algorithm adds scheme states, scheme transitions or both to the existing transformation scheme according to the construction rules. The following paragraphs will describe the construction with an example which is presented in the following listing. The implementation of the construction algorithm will be discussed in Chapter 8.

Listing 6.17: Description of a Construction Example in TSDL.

```
1 (
2   (
3     < ti > {C1} |
4     < tj >
5   ) [0 .. 2] {C2},
6   <{mC1}>
7 ) {C3}
```

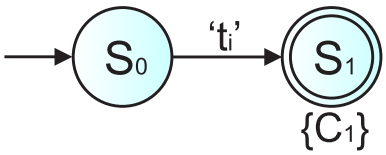
The transformation scheme description consists of an outer subscheme where the constraint C_3 has to be satisfied after its application. This outer subscheme embeds an inner subscheme which can be applied zero to two times and a $\mathcal{META}\text{Constraint}$ mC_1. The inner subscheme and the $\mathcal{META}\text{Constraint}$ are combined via a sequence where the constraint C_2 has to be satisfied after each application of the inner subscheme. Furthermore, the inner subscheme contains two FermaT transformations t_i and t_j which are combined via an alternative where the constraint C_1 has to be satisfied after each application of the first transformation t_i.

Step 1

The transformation scheme description starts with a left bracket followed by another left bracket. This indicates the described nested subschemes. The corresponding constructions will be executed when each of the subschemes ends which in turn is indicated by

a right bracket. The following item is the non-terminal *transformation* t_i. This item is extended by a constraint C_1 and causes the execution of a basic construction. After the execution, a construct has been created which consists of the scheme states S_0 and S_1 and the FermaT transformation t_i. The state S_0 is the initial scheme state whereas the state S_1 is the final scheme state. This final scheme state has to satisfy the constraint C_1. Figure 6.12 shows the resulting basic construct.

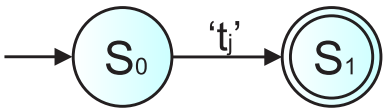
Figure 6.12: Construction Example Step 1.



Step 2

The first basic construct has been created and the next item of the transformation scheme description will be processed. This item is the non-terminal *alternative* which causes the execution of an alternative construction. As mentioned before, this construction combines two existing transformation schemes. The first transformation scheme of this combination is the basic construct which has been created in Step 1. The second transformation scheme does not exist yet. Therefore, the second scheme has to be created before the alternative construction can be executed. The next item after the alternative in the transformation scheme description is the non-terminal *transformation* t_j. It causes the execution of another basic construction which consists of the scheme states S_0 and S_1 and the FermaT transformation t_j. The state S_0 is the initial scheme state whereas the state S_1 is the final scheme state. The result of this construction is shown in Figure 6.13.

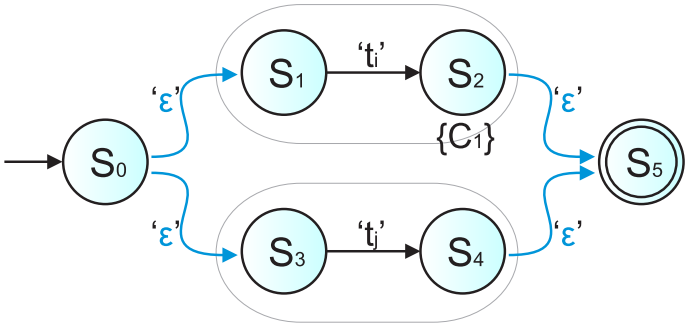
Figure 6.13: Construction Example Step 2.



Step 3

At this point, two basic constructs have been created and the alternative construction which has been mentioned in Step 2 can be executed. This construction creates a new construct which combines both existing basic constructs. A new initial scheme state S_0 will be created whereas the existing initial scheme states become common ones. The new initial scheme state S_0 will be connected via two new created ε-transitions with the first scheme state of each existing transformation scheme. Furthermore, a new final scheme state S_5 will be created whereas the existing final scheme states become common ones as well. Each of the last scheme states of the existing transformation schemes will be connected via an ε-transition with the new final scheme state S_5. However, the constraint C_1 remains assigned to the same scheme state as before. After the construction, the scheme states of the existing basic constructs have to be renamed to guarantee a consistent numbering. Figure 6.14 shows the resulting alternative construct.

Figure 6.14: Construction Example Step 3.

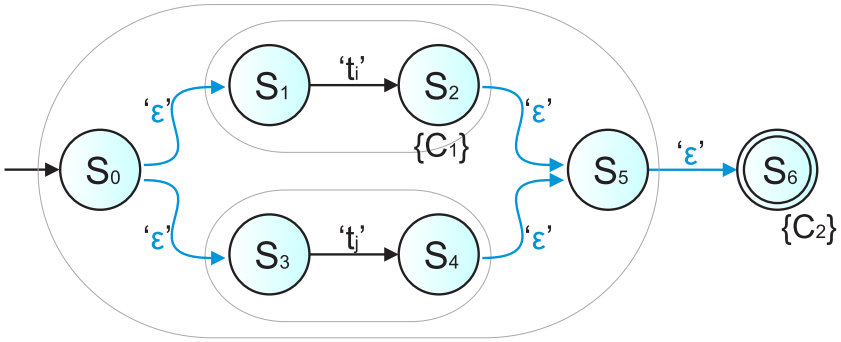


Step 4

The alternative construct has been created and the next item of the transformation scheme description is a right bracket. This indicates the end of the inner subscheme which has been started in Step 1. For this reason, a subset construction will be executed. This construction causes an extension of the existing alternative construct by an ε-transition and a scheme state S_6. The created scheme state S_6 in turn will be the final scheme state of the subscheme construct whereas the final scheme state S_5 of the existing transformation scheme will be converted into a common one. The right bracket in the transfor-

mation scheme description is followed by a non-terminal *quantifier* and a non-terminal *constraint*. The constraint C_2 will be added directly to the created scheme state S_6 of the extended construct whereas the quantifier causes the execution of a quantifier construction. Figure 6.15 shows the resulting subset construct.

Figure 6.15: Construction Example Step 4.



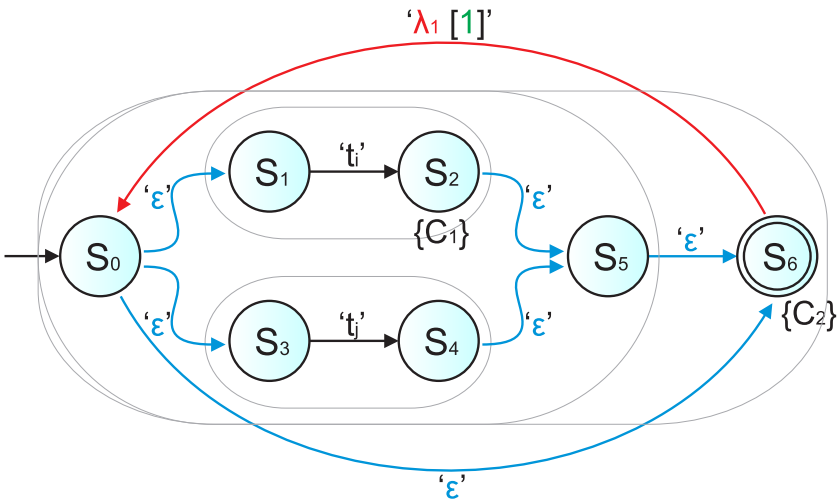
Step 5

The given quantifier in the transformation scheme description defines that the subscheme can be applied zero to two times. Therefore, it surrounds the subscheme construct by a quantifier construct of type III. This is because the given endpoints q_0 and q_n of the non-terminal *quantifier* define the construct to be a combination of option and repeat construct in terms of the regular Thompson Construction. For this reason, two scheme transitions will be added to the existing transformation scheme. The first scheme transition is an ε-transition which connects the initial scheme state S_0 with the final scheme state S_6. The second transition is a λ-transition which connects the final scheme state S_6 with the initial scheme state S_0. Figure 6.16 shows the resulting quantifier construct.

Step 6

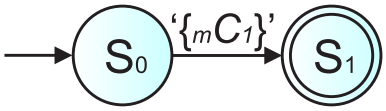
The quantifier construct has been created and the next item of the transformation scheme description will be processed. This item is the non-terminal *sequence* which causes the execution of a sequence construction. As well as the alternative construction, this construction combines two existing transformation schemes. The first transformation scheme

Figure 6.16: Construction Example Step 5.



of this combination is the quantifier construct which has been created in Step 5. The second transformation scheme does not exist yet. Therefore, the second scheme has to be created before the sequence construction can be executed. The next item after the sequence in the transformation scheme description is the non-terminal *transformation* mC_1. It causes the execution of another basic construction which consists of the scheme states S_0 and S_1 and the $\mathcal{META}$Constraint mC_1. The state S_0 is the initial scheme state whereas the state S_1 is the final scheme state. The result of this construction is shown in Figure 6.17.

Figure 6.17: Construction Example Step 6.

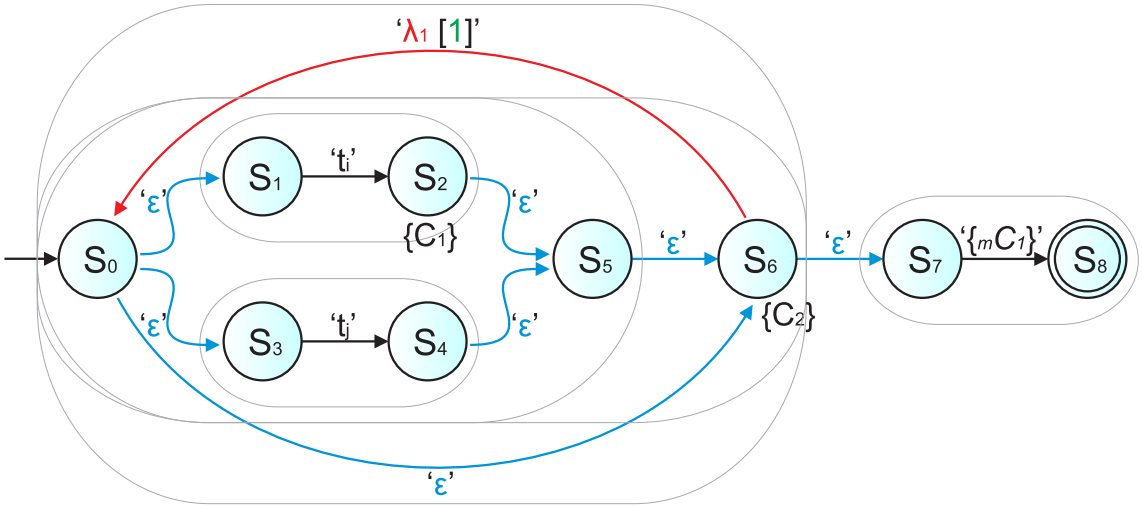


Step 7

At this point, a quantifier construct and a basic construct have been created and the sequence construction which has been mentioned in Step 6 can be executed. This construction creates a new construct which combines both existing transformation schemes. A

new ε-transition will be created which connects the last scheme state S_6 of the existing quantifier construct with the first scheme state of the existing basic construct. As a result, the final scheme state S_6 of the quantifier construct and the initial scheme state of the basic construct become common scheme states. However, the constraints C_1 and C_2 remain assigned to the same scheme states as before. After the construction, the scheme states of the existing basic construct have to be renamed to guarantee a consistent numbering. Figure 6.18 shows the resulting sequence construct.

Figure 6.18: Construction Example Step 7.

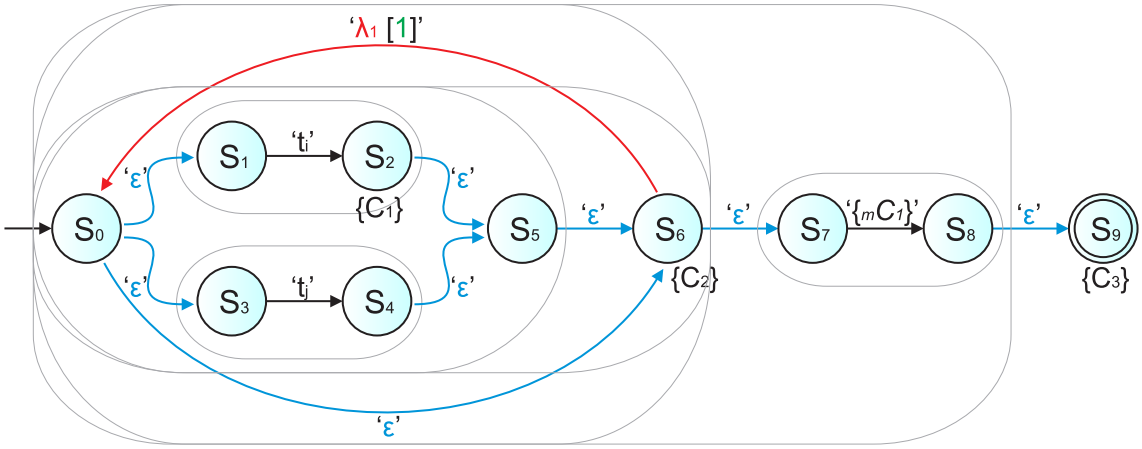


Step 8

The sequence construct has been created and the next item of the transformation scheme description is another right bracket. This indicates the end of the outer subscheme which has been started in Step 1. For this reason, another subset construction will be executed. As well as in Step 4, this construction causes an extension of the existing sequence construct by an ε-transition and a scheme state S_9. The created scheme state S_9 in turn will be the final scheme state of the subscheme construct whereas the final scheme state S_8 of the existing transformation scheme will be converted into a common one. The right bracket in the transformation scheme description is followed by a non-terminal *constraint*. Therefore, the constraint C_3 will be added directly to the created scheme state S_9 of the extended construct. Figure 6.19 shows the resulting subset construct. Since there are no more items

in the transformation scheme description, the construction has been finished at this point.

Figure 6.19: Construction Example Step 8.



6.5 Conversion of Transformation Schemes

As mentioned before, the presented approach uses a slightly adapted version of the Thompson Construction which converts the TSDL code into an equivalent transformation scheme. This scheme is based on an ε-NFA which has a relatively structured appearance [2]. In general, ε-NFA based transformation schemes can be beneficial in terms of comprehension and visualisation purposes. For example, the layout of the included scheme states and scheme transitions can be created on the basis of the constructs which have been discussed in Section 6.4.

However, the problem with ε-NFA based transformation schemes is that the transition function of these schemes is defined as function from an individual scheme state S_i to a powerset of this state $P(S_i)$. In other words, they are working with a set of scheme states rather than a single scheme state. This causes problems during further processing especially in regard to execution speed. Furthermore, it increases the complexity of algorithms which processes such a transformation scheme [40]. For this reason, it can be beneficial to convert a given ε-NFA transformation scheme into a DFA based transformation scheme.

On the other hand, the conversion might cause a massive increase of the number of

scheme states within the transformation scheme. For example, if an ε-NFA contains n scheme states, the converted DFA can contain up to 2^n scheme states [1]. However, the advantages of DFA based transformation schemes normally outweigh the disadvantages. Therefore, the further processing which is described in Chapter 7 is adapted for DFA based transformation schemes. The following paragraphs will describe the conversion on the basis of the construction example of Section 6.4.

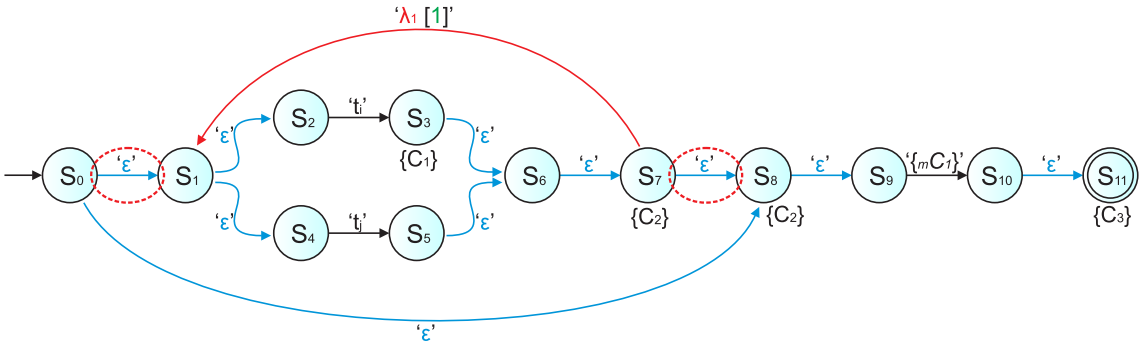
6.5.1 Preparation for the Conversion

Before the actual conversion can be executed, the transformation scheme has to be prepared. This preparation is split into two steps which will be discussed in the following paragraphs.

Expansion of Transformation Schemes

The first step of the preparation expands quantifier constructs of the type III which leads to advantages while further processing of the constructed scheme. More precisely, the expansion adds two ε-transitions and two scheme states to the existing transformation scheme. Figure 6.20 shows the finished construction example of Section 6.4 after the type III quantifier construct has been expanded. The implementation of the preparation algorithm will be discussed in Chapter 8.

Figure 6.20: ε-NFA based Transformation Scheme after the Expansion.



After the expansion, the creation of the search space will be simplified because the definition of redundant transformation sequences have been reduced. Moreover, the sequences within the search space have been reduced as well. In fact, a lot of commonly

useless transformation sequences have been eliminated. The following listing shows the defined search space of the unexpanded transformation scheme. In this listing, the constraints C_1, C_2 and C_3 are surrounded by curly brackets and indicate where they have to be satisfied in a particular transformation sequence. Furthermore, the included $\mathcal{META}$Constraint mC_1 has not been decomposed. This decomposition will be discussed in Chapter 7.

Listing 6.18: Defined Search Space of the Unexpanded Transformation Scheme.

```
1  Sequence 1: {C2}, {C2}, {mC1}, {C3}
2  Sequence 2: {C2}, ti, {C1}, {C2}, {mC1}, {C3}
3  Sequence 3: {C2}, tj, {C2}, {mC1}, {C3}
4  Sequence 4: {C2}, {mC1}, {C3}
5  Sequence 5: ti, {C1}, {C2}, {C2}, {mC1}, {C3}
6  Sequence 6: ti, {C1}, {C2}, ti, {C1}, {C2}, {mC1}, {C3}
7  Sequence 7: ti, {C1}, {C2}, tj, {C2}, {mC1}, {C3}
8  Sequence 8: ti, {C1}, {C2}, {mC1}, {C3}
9  Sequence 9: tj, {C2}, {C2}, {mC1}, {C3}
10 Sequence 10: tj, {C2}, ti, {C1}, {C2}, {mC1}, {C3}
11 Sequence 11: tj, {C2}, tj, {C2}, {mC1}, {C3}
12 Sequence 12: tj, {C2}, {mC1}, {C3}
```

There are twelve transformation sequences in the search space. A lot of them are useless. For example, the first and the fourth transformation sequence are equivalent in practical terms. Furthermore, there are four transformation sequences which start with constraint C_2. Three of them, in particular the second, the third and the fourth sequence, are violating the intention of the transformation scheme description. The constraint C_2 has to be satisfied after the application of the transformation t_i or the transformation t_j rather than before. However, the first sequence is not violating the intention of the description because it defines that an application of transformation t_i or the transformation t_j can be skipped if the constraint C_2 has already been satisfied. The following listing shows the defined search space of the expanded transformation scheme. In this listing, the constraints C_1, C_2 and C_3 are surrounded by curly brackets and indicate where they have to be satisfied in a particular transformation sequence. Furthermore, the included $\mathcal{META}$Constraint mC_1 has not been decomposed.

Listing 6.19: Defined Search Space of the Expanded Transformation Scheme.

```
1 Sequence 1: {C2}, {mC1}, {C3}
2 Seqeunce 2: ti, {C1}, {C2}, ti, {C1}, {C2}, {C2}, {mC1}, {C3}
3 Seqeunce 3: ti, {C1}, {C2}, tj, {C2}, {C2}, {mC1}, {C3}
4 Seqeunce 4: ti, {C1}, {C2}, {C2}, {mC1}, {C3}
5 Seqeunce 5: tj, {C2}, ti, {C1}, {C2}, {C2}, {mC1}, {C3}
6 Seqeunce 6: tj, {C2}, tj, {C2}, {C2}, {mC1}, {C3}
7 Seqeunce 7: tj, {C2}, {C2}, {mC1}, {C3}
```

There are only seven transformation sequences in the search space. None of these sequences is useless or violates the intention of the transformation scheme description. However, some constraints may appear twice in a row which is a result of the expansion. This problem can be solved easily after the generation of the search space.

Reassignment of Constraints

The second step of the preparation reassigns all constraints from the scheme states to the incoming scheme transitions of these states. Due to the construction algorithm, it is not possible that the incoming scheme transition is a λ-transition. If the incoming scheme transition is a FermaT transformation, the constraint will simply be appended. However, if the incoming scheme transition is an ε-transition, the transition will be converted into a λ-transition because its usage depends on a particular condition which is the reassigned constraint in this case.

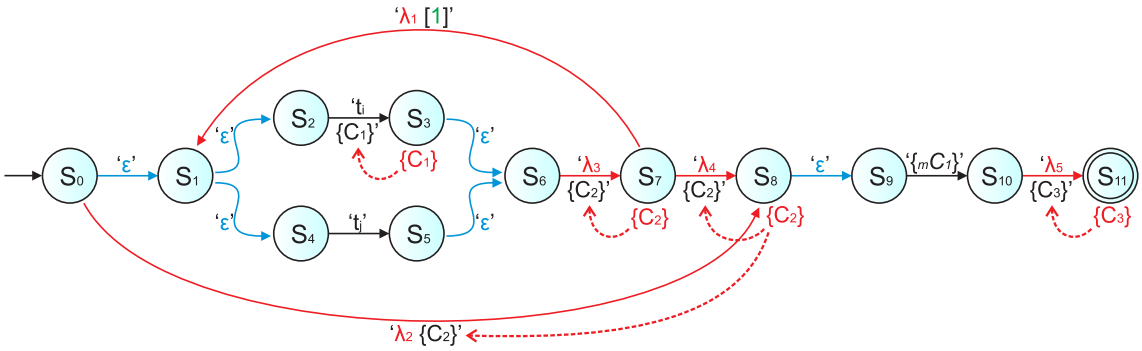
The reassignment of constraints is necessary because constraints which are included into scheme states cannot be processed with common conversion techniques like the powerset construction [1]. For example, it is possible that the ε-closure or the move method which are used during the conversion combine several ε-NFA scheme states to a single DFA scheme state. The result of this combination is that the assignment of constraints to particular scheme states within the DFA based transformation scheme may become inconsistent.

However, the reassignment process as such does not cause any problems because the only scheme state which is reachable without using a scheme transition is the initial one. All other scheme states within a transformation scheme are only reachable via incoming

scheme transitions. Due to the construction algorithm, it is impossible that the initial scheme state has to satisfy a constraint.

As a matter of fact, the reassignment has no practical effect on the processing of the transformation scheme. It only defines that a constraint has to be satisfied after a scheme transition has been used instead of that a constraint has to be satisfied at a particular scheme state. After the reassignment process, the preparation for the following conversion has been completed. Figure 6.21 shows the finished construction example of Section 6.4 after the type III quantifier construct has been expanded and after the given constraints have been reassigned.

Figure 6.21: ε-NFA based Transformation Scheme prepared for the Conversion.



6.5.2 Conversion via the Powerset Construction

After the preparation, the actual conversion will be realised via a powerset construction. This is widely known as one of the standard techniques to convert an ε-NFA into a DFA which recognises the same formal language [1]. The powerset construction is based on the idea that a set of states T within an ε-NFA N is a single state S_i within a DFA D. Therefore, a state table D_{stat} and a transition table D_{tran} will be introduced [17]. These tables store the set of states T which can be reached for every symbol of the alphabet. Furthermore, they store the symbols themselves which connect the states [2]. The implementation of the powerset construction will be discussed in Chapter 8. The following methods are used to generate the state table D_{stat} and the transition table D_{tran}:

1. ε-closure(s_i) - Returns a set of states T which are reachable from a single state S_i via ε-transitions.

2. ε-closure(T) - Returns a set of states T' which are reachable from a set of states T via ε-transitions.

3. move(T, a) - Returns a set of states T' which are reachable from a set of states T via the input symbol a.

These are the standard methods which are used in combination with the powerset construction. The ε-closure(S_i) method has to be used on the initial scheme state of a ε-NFA based transformation scheme. It returns the first set of scheme states T which represents the initial scheme state of the created DFA based scheme. Afterwards, the move(T, a) and the ε-closure(T) methods have to be used alternately to create the state table D_{stat} and the transition table D_{tran}. The implementation of these method will be discussed in Chapter 8. The following table shows the state table D_{stat}.

Table 6.2: State Table D_{stat} of the prepared ε-NFA based Transformation Scheme.

D_{stat}	Set of ε-NFA States	Contains Final State
S_0	S_0, S_1, S_2, S_4	No
S_1	S_8, S_9	No
S_2	S_3, S_6	No
S_3	S_5, S_6	No
S_4	S_{10}	No
S_5	S_7	No
S_6	S_{11}	No
S_7	S_1, S_2, S_4	Yes

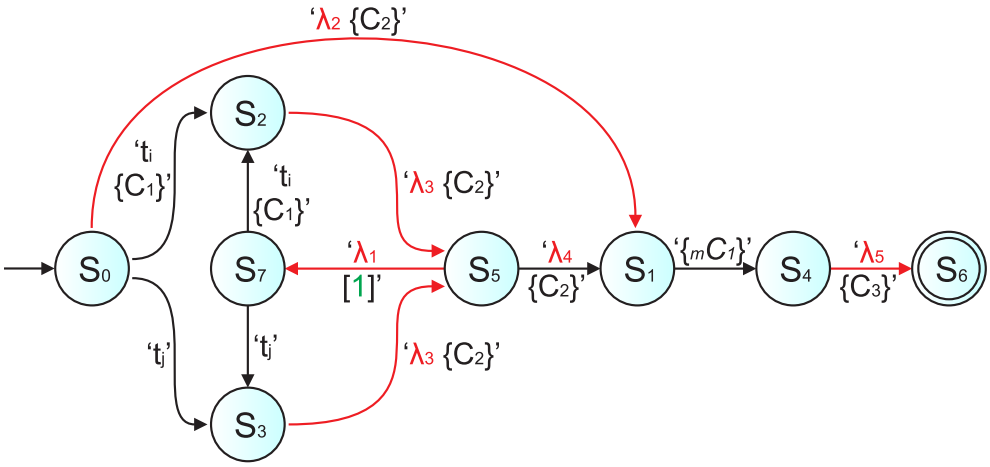
The resulting DFA based transformation scheme contains eight scheme states. The state S_0 is the only initial scheme state whereas the state S_6 is a final scheme state. However, the created DFA based transformation scheme will contain less scheme states than the original ε-NFA based transformation scheme. As mentioned before, this is often but not always the case. The following table shows the transition table D_{tran}.

Table 6.3: Transition Table D_{stat} of the prepared ε-NFA based Transformation Scheme.

From D_{stat}	To D_{stat}	Via D_{tran}
S_0	S_1	λ_2 $\{C_2\}$
S_0	S_2	t_i $\{C_1\}$
S_0	S_3	t_j
S_1	S_4	$\{mC_1\}$
S_2	S_5	λ_3 $\{C_2\}$
S_3	S_5	λ_3 $\{C_2\}$
S_4	S_6	λ_5 $\{C_3\}$
S_5	S_1	λ_4 $\{C_2\}$
S_5	S_7	λ_4 $[1]$

The resulting DFA based transformation scheme contains eight scheme transitions. The ε-transitions have been removed and there is no scheme state which has two or more outgoing scheme transitions with the same name. The assembled DFA based transformation scheme is presented in Figure 6.22.

Figure 6.22: DFA based Transformation Scheme after the Conversion.



6.6 Summary

The presented chapter has described a transformation scheme approach which is based on automata theory and which can be used to model an entire program transformation process. It has discussed a formal language called TSDL which provides an interface to describe transformation schemes and it has demonstrated a slightly adapted version of the Thompson Construction for an automated creation of ε-NFA based transformation schemes from TSDL. The use of transformation capabilities and effects to define transformation sets with the aid of so-called $\mathcal{META}$Constraints has been explained and the conversion of ε-NFA based transformation schemes into DFA based transformation schemes has been demonstrated.

Chapter 7

Prediction and Search Based Constraint Satisfaction

Objectives

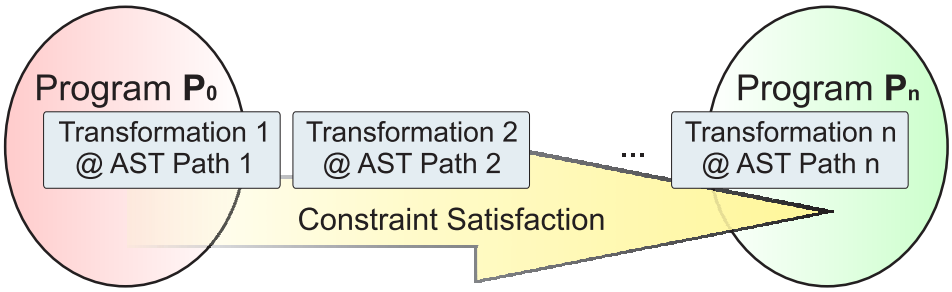
- To describe the use of transformation capabilities and effects.
- To discuss prediction techniques for an improved transformation sequence search.
- To describe various search tactics in combination with transformation schemes.

A program transformation process based on the FTE can be considered as the application of a sequence of FermaT transformations. During this process, each individual transformation will be applied on a particular AST path [31]. Furthermore, each transformation has been mathematically proven to preserve the denotational semantics of the initial program P [83]. Therefore, we can assume that the initial program P_0 is semantically equivalent to the resulting program P_n and to all intermediate program states P_i which are generated during the program transformation process. To put it briefly, a program P_n is proven to be semantically equivalent to the program P if it has been modified via FermaT transformations without exception.

As described in Chapter 4, the target of a program transformation can be defined via constraints. Moreover, transformation schemes are used to model the transformation process. A maintainer who creates such transformation schemes assumes that each of them includes one or more applicable transformation sequences which satisfy the respective constraints. Furthermore, these schemes can be described by a formal language which has been discussed in Chapter 6.

The presented approach provides the possibility to outline the process of program transformation. This in turn leads to a search problem where different algorithms are used to search for transformation sequences consisting of FermaT transformations. The application of these sequences on particular AST paths within the given program must lead to a satisfaction of the respective constraints. This process is shown in Figure 7.1.

Figure 7.1: Constraint Satisfaction via Program Transformation.



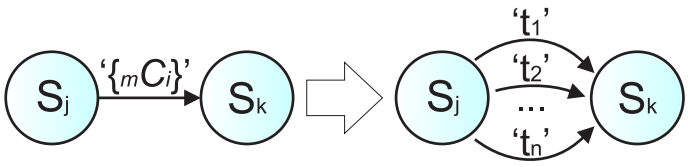
The following sections discuss a prediction technique to filter inapplicable transformation sequences as well as sequence evaluation technique. These techniques are based on transformation capabilities and effects which have been discussed in Chapter 5. Furthermore, a particular search tactic will be discussed to discover transformation sequences whose application satisfy the given constraints.

7.1 Dealing with Transformation Sets

As discussed in Chapter 6, the proposed approach of transformation schemes supports the definition of transformation sets via so-called $\mathcal{META}$Constraints. These have to be decomposed at some point in the constraint based program transformation process. In the

proposed approach, the decomposition is done before the search space will be created. Figure 7.2 shows the process of transformation set decomposition.

Figure 7.2: Decomposition of a Transformation Set.



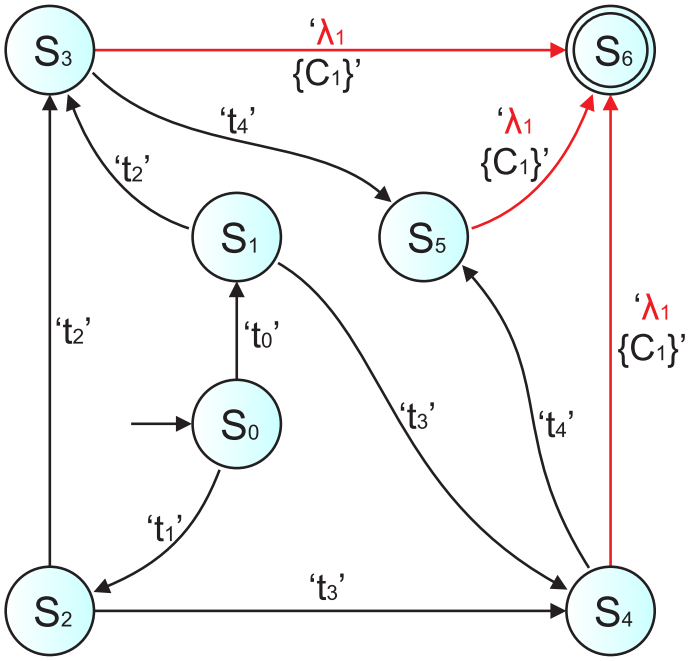
The decomposition process replaces a $\mathcal{META}$Constraint transition by a set of transitions which is defined by the $\mathcal{META}$Constraint. Each of the transitions within the set has the same source and target state and will be applied on the AST path defined by the $\mathcal{META}$Constraint transition. In other words, the decomposition process creates an alternative between the transformations that can be applied on a determined AST path.

7.2 Creation of the Search Space

The creation of the search space from a given transformation scheme is an important process. In fact, a transformation scheme defines whether a given transformation sequence is accepted or not. Only if it is accepted, it is in the search space for the respective transformation sequence search. However, it is a difficult and inefficient process to generate a search space which contains all possible transformation sequences of the given set of transformations Σ in advance and filter those sequences which are not accepted afterwards. For this reason, an algorithm has been developed which directly generates the set of acceptable sequences and therefore the search space. This algorithm will be discussed in Chapter 8. It generates all accepted transformation sequences of a DFA based transformation scheme. The result of the algorithm is a set of transformation sequences which can be considered as search space for transformation sequence searches. Figure 7.3 shows an example of a transformation scheme which will be used in the following to demonstrate the generation of such a search space.

The transformation scheme is DFA based and consists of seven states and eleven transitions of which eight are transformations and three are λ-transitions. The transformations

Figure 7.3: Transformation Scheme to Demonstrate the Generation of a Search Space.



are black coloured whereas the λ-transitions are red coloured. The presented transformation scheme is able to generate transformation sequences which consists of at least two and at most three transformations. Moreover, it defines that there is no path which reaches the final state S_6 without passing the constraint C_1.

The algorithm starts at the initial state S_0. This state is not accepted and has two outgoing transitions. The first one is the transition t_0 which leads to the state S_1. Because it obviously is a transformation, it will be added as first element to the current sequence. The state S_1 is not accepted and has two outgoing transitions as well. The first one is the transition t_2 which leads to the state S_3. Again, it is a transformation and therefore will be added as second element to the current sequence. As well as the first two visited states, the state S_3 is not accepted and has two outgoing transitions. The first one is the transition t_4 which leads to the state S_5. Once more, it is a transformation and therefore will be added as third element to the current sequence. The state S_5 is not accepted and has one outgoing transition which leads to the state S_6. This transition is a λ-transition with an appended constraint C_1 which will be added as fourth element to the current sequence. The state S_6 in turn is an accepted state. For this reason, the current sequence which

consists of the elements t_0, t_2, t_4 and C_1 will be added to the search space. Furthermore, the state has no outgoing transitions which means that the algorithm has to perform a backtrack. This leads to the state S_3 because it has another outgoing transition which can be traversed. It is a λ-transition which leads to the accepted state S_6. Again, the current sequence which consists of the elements t_0, t_2 and C_1 will be added to the search space and the algorithm has to perform a backtrack. The algorithm will be executed until all possible paths of the transformation scheme which lead to the accepted state have been traversed. The following Listing shows the created search space which is sorted by the transformation sequence length.

Listing 7.1: Generated Search Space.

```
1 Sequence 1 { length = 3 } : t0, t2, {C1}
2 Sequence 2 { length = 3 } : t0, t3, {C1}
3 Sequence 3 { length = 3 } : t1, t2, {C1}
4 Sequence 4 { length = 3 } : t1, t3, {C1}
5 Sequence 5 { length = 4 } : t0, t2, t4, {C1}
6 Sequence 6 { length = 4 } : t0, t3, t4, {C1}
7 Sequence 7 { length = 4 } : t1, t2, t4, {C1}
8 Sequence 8 { length = 4 } : t1, t3, t4, {C1}
```

As shown in the listing, the constraint is an element of the transformation sequence. It is not changing the program but it determines the point in the sequence where a particular transformation target has to be achieved. However, the generated search space it very small. It only contains eight very short sequences. In contrast, the generated search space often contains thousands of sequences which will be presented in Chapter 9.

7.3 Applicability Prediction of Transformation Sequences

As discussed in Chapter 2, there exist a transformation step prediction approach which tries to predict transformation sequences that satisfy a given target. On the one hand, this technique solves some of the existing problems while applying a program transformation process. On the other hand, other problems have been introduced which have also been discussed in Chapter 2. Therefore, the approach presented in this thesis is mainly based on a particular search tactic which will be discussed in Section 7.6. Nevertheless, it is possible to use prediction techniques to support these tactics. In the case of the pro-

posed approach of this thesis, a technique is used which tries to predict whether an entire transformation sequence is applicable or not. If it is not applicable, the sequence can be removed from the search space.

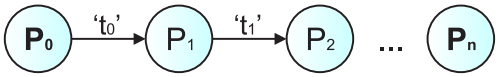
As shown in Listing 5.1, the applicability of a transformation is determined within the first procedure of the transformation itself. This is called the applicability condition of a transformation. It analyses the program on which the respective transformation is supposed to be applied and returns either *Pass* which means the code has a valid structure or *Fail* which means the structure of the code is invalid.

The proposed technique attempts to predict the applicability of a transformation on a given program. Therefore, it only takes the transformation and not the application path into consideration. Only if all transformations within a sequence have been predicted to be applicable, the entire transformation sequence can be considered as applicable. As mentioned before, applicable transformation sequences have to remain in the search space whereas inapplicable transformation sequences can be removed. The fact that the prediction technique uses a mathematical model which does not include the AST path at which a transformation has to be applied leads to a simplified application of the sequence. In general, the application of a transformation sequence behaves like a tree if any of the transformations within the sequence has to be applied on an AST path which is not definite and therefore indetermined. In this tree, the nodes are the program states where the initial program is the root and the final program is a leaf. This will be discussed in Section 7.6 in detail. On the other hand, the application of a simplified transformation sequence which does not consider AST paths behaves like a list where each program state is an element of the list that has at most one successor.

A simplified transformation sequence is a model of a transformation sequence which cannot be applied. The mathematical model uses the simplified transformation sequence to predict the applicability of a particular transformation. Figure 7.4 shows how an application of a transformation sequence is modelled on the basis of a simplified transformation sequence. The state P_0 of this example represents the initial program whereas the state P_n represents the final program.

The mathematical model of the prediction technique uses the capabilities of a trans-

Figure 7.4: Modelled Application of a Transformation Sequence.



formation to calculate if the AST types which are needed for an application of this transformation are in the set of AST types of the respective program state. More precisely, it uses the set of AST types of a given program as well as three of the five defined capability sets. The mathematical model does not consider the number of types which are in the capability sets or in the sets of the respective program states. This is because the set C_P does not contain the number of AST types which can possibly be created because of dependencies to the given program. For example, the transformation Reduce Nots is able to generate a random number of various AST types depending on the given AST types within the program which has to be transformed. There are no theoretical limits of this number whereas practical limits are determined by the limitation of the environment in terms of hardware and software.

As mentioned before, the prediction technique uses a mathematical model which is based on sets. One of these sets contains the AST types of a given program. Therefore, a function $s(P_i)$ will be introduced where P_i is a particular program state and Λ is the set of all existing WSL AST types which will be presented in Appendix A. This function is defined as following:

$$s : P_i \longmapsto E \subseteq \Lambda$$

The function $s(P_i)$ returns a set of AST types E which is a subset of the set of all existing AST types Λ. For example, let P_0 be a program which consists of only one statement that assigns a constant to a variable. The function $s(P_i)$ returns a set which contains the general type T_Assign, the group type T_Statements and the specific types T_Assignment, T_Number and T_Var_Lvalue for the program P_0 as input element.

In addition to the set of AST types of a given program, the mathematical model also uses capabilities of constraints. One of the sets of capabilities which is used is the set A. This has been discussed in Chapter 5. It contains the AST types on which a transformation

is applicable. A function $a(t_i)$ will be introduced where t_i is a particular transformation and Λ is the set of all existing WSL AST types. This function is defined as following:

$$a : t_i \longmapsto A \subseteq \Lambda$$

The function $a(t_i)$ returns a set of AST types A which is a subset of the set of all existing AST types Λ. For example, the function returns a set which only contains the specific type T_While for the While to Floop transformation as input element whereas it would return a set which only contains the specific type T_Floop for the Floop to While as input element.

Another capability set which is used by the mathematical model is the set C_C. This has been discussed in Chapter 5 as well. It contains the AST types which will certainly be created after a transformation has been applied. A function $c_c(t_i)$ will be introduced where t_i is a particular transformation and Λ is the set of all existing WSL AST types. This function is defined as following:

$$c_c : t_i \longmapsto C_C \subseteq \Lambda$$

The function $c_c(t_i)$ returns a set of AST types C_c which is a subset of the set of all existing AST types Λ. For example, the function returns a set which contains the general type T_Guarded, the group type T_Statements and the specific types T_Cond, T_Floop and T_Exit for the While to Floop transformation as input element.

The last capability set which is used by the mathematical model is the set C_P. As well as the other sets of capabilities, this has been discussed in Chapter 5. It contains the AST types which will possibly be created after a transformation has been applied. A function $c_p(t_i)$ will be introduced where t_i is a particular transformation and Λ is the set of all existing WSL AST types. This function is defined as following:

$$c_p : t_i \longmapsto C_P \subseteq \Lambda$$

The function $c_p(t_i)$ returns a set of AST types C_p which is a subset of the set of all existing AST types Λ. For example, the function returns a set which contains the spe-

cific types T_And, T_Equal, T_Even, T_False, T_Greater, T_Greater_Eq, T_In, T_Less, T_Less_Eq, T_Not, T_Not_Eq, T_Not_In, T_Odd, T_Or and T_True for the While to Floop transformation as input element.

In the presented mathematical model, a program state P_i which has been generated through the application of transformation t_{i-1} on the program state P_{i-1} consists of a set of AST types E. This set is accessed by the function $s(P_i)$. Furthermore, a transformation t_i consists of three sets A, C_C and C_P which are accessed on the basis of the three described functions $a(t_i)$, $c_c(t_i)$ and $c_p(t_i)$. The application of a transformation t_i in terms of the mathematical model is defined as union of the transformation capability sets C_C and C_P and the set of AST types E of the program state P_i on which the transformation is applied. The result is the set of AST types W which are in the generated program state P_{i+1} in the worst case:

$$W = s(P_i) \cup c_c(t_i) \cup c_p(t_i)$$

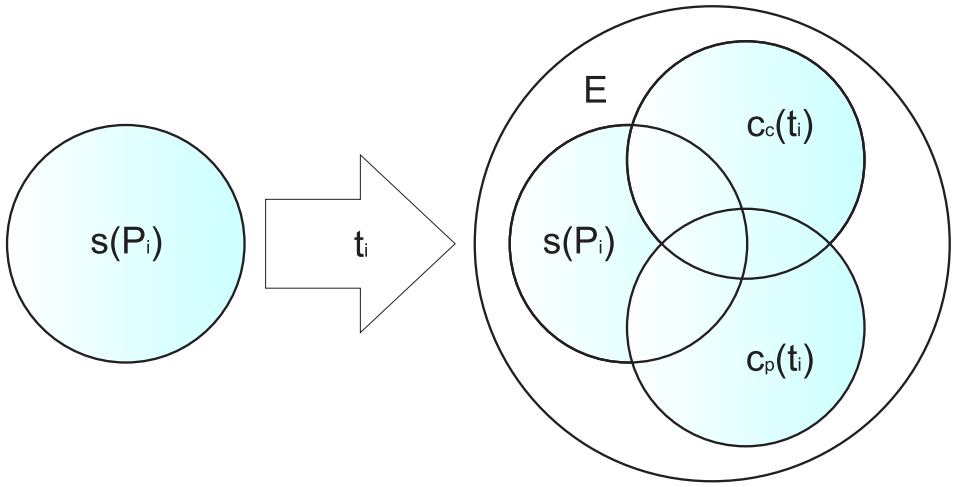
The presented formula attempts to predict the AST types which can be found within the following program state on the basis of a given program state and the capabilities of the applied transformation. Figure 7.5 shows the application of a transformation t_i in terms of the mathematical model with the aid of a set diagram.

On the basis of this formula, it is possible to derive another formula which allows to predict the set of AST types which are in each program state in the worst case. Within a given transformation sequence, the set of AST types W_j of a program state P_j can be predicted by a union of the set of AST types of the initial program P_0 and the capability sets C_C and C_P of each transformation which needs to be applied to generate the program state P_j:

$$W_j = s(P_0) \cup \bigcup_{i=0}^{j-1} c_c(t_i) \cup \bigcup_{i=0}^{j-1} c_p(t_i)$$

To check whether an entire transformation sequence is valid or not, the set of AST types of each state has to be calculated on the basis of the presented formula. Based on the resulting sets, the entire transformation sequence is applicable if and only if the following formula is valid:

Figure 7.5: Set Diagram of the Application of a Transformation in terms of the Mathematical Model.



$$\forall i : \begin{cases} a(t_i) \cap s(P_i) \neq \emptyset, & i = 0 \\ a(t_i) \cap W_j \neq \emptyset, & i > 0 \end{cases}$$

In a lot of cases, the presented prediction technique is able to identify an inapplicable transformation sequence. This can be used to reduce the search space for a subsequent search. However, the technique has its limitations as well. It only predicts which AST types are in a program state in the worst case. Moreover, it only considers if the AST type on which a transformation can be applied is in the set of AST types of the respective program state rather than to take the entire applicability condition of a transformation into account. This will be explained with an example which is based on the following transformation sequence.

Listing 7.2: Example Transformation Sequence T_S to Demonstrate Applicability Prediction.

```
1 < Else If to Elsif @ /0/ >,
2 < While to Floop @ /0,1,1,2/ >,
3 < For to While @ /0/ >,
4 < Insert Assertions @ /0/ >
```

The transformation sequence consists of only four transformations which are the Else If to Elsif, the While to Floop, the For to While and the Insert Assertions. All of them will

be applied on the AST path /0/ except the second transformation which will be applied on the AST path /0,1,1,2/. The following listing shows a WSL example program which will be used to demonstrate an applicability prediction of the presented transformation sequence.

Listing 7.3: Initial Example Program P_0 to Demonstrate the Prediction of Transformation Sequence Applicability.

```
1  IF i = 0 THEN
2    j := 10;
3    k := 20
4  ELSE
5    IF i = 1 THEN
6      j := 0;
7      k := 0;
8      WHILE j < 100 DO
9        j := j + 1;
10       k := k + j
11     OD
12   FI
13 FI
```

The example program represents the initial program P_0. It contains an *IF* statement which surrounds some assignments as well as a *WHILE* loop. The following Table shows the set of AST types $s(P_0)$ of which the program consists. Furthermore, it shows the predicted set of AST types W_1 and the set of AST types $s(P_1)$. The set W_1 has been created on the basis of the described prediction technique whereas the set $s(P_1)$ has been extracted from the program state S_1. This set serves only to compare the precision of the prediction technique because the program state S_1 would not exist in real applications.

Table 7.1: Set of AST Types $s(P_i)$ of Program P_0 and P_1 and Predicted Set of AST Types W_j of Program P_1.

$s(P_0)$	W_1	$s(P_1)$
T_Assign	T_Assign	T_Assign
		Continued on next page

AST types - continued from previous page.

$s(P_0)$	W_1	$s(P_1)$
T_Assignment	T_Assignment	T_Assignment
T_Cond	T_Cond	T_Cond
T_Equal	T_Equal	T_Equal
T_Guarded	T_Guarded	T_Guarded
T_Less	T_Less	T_Less
T_Number	T_Number	T_Number
T_Plus	T_Plus	T_Plus
T_Statements	T_Statements	T_Statements
T_True	T_True	
T_Variable	T_Variable	T_Variable
T_Var_Lvalue	T_Var_Lvalue	T_Var_Lvalue
T_While	T_While	T_While

The table compares the set of AST types of the initial program P_0 with the predicted set of AST types W_j and the real set of AST types $s(P_i)$ of the program state P_1. The state P_1 is the result of applying the first transformation t_0 of the example transformation sequence on the initial program P_0. As shown in the Table, the transformation t_0 does not introduce any AST types but it removes one which has not been noticed by the prediction algorithm. This is a general problem of the mathematical model on which the prediction technique is based. It is not able to detect the removal of AST types from the respective program.

To predict whether the transformation sequence is applicable or not, the presented formula has to check if the result of the intersection of the sets $s(P_0)$ and $a(t_0)$ is an empty set. This is not the case because the set contains the specific type T_Cond which is the only AST type in the set $a(t_0)$. Therefore, it is necessary to check another intersection which is of the sets W_1 and $a(t_1)$. The result of this intersection is also not an empty set because it contains the specific type T_While which is the only AST type in the set $a(t_1)$. To put it briefly, the first transformation which is the Else If to Elsif as well as the second transformation which is the While to Floop have been predicted to be applicable which is correct in both cases. The following listing shows the program state P_1 which is the result

of applying transformation t_0 on the initial program P_0.

Listing 7.4: Example Program P_1 to Demonstrate the Prediction of Transformation Sequence Applicability.

```
1  IF  i = 0 THEN
2     j := 10;
3     k := 20
4  ELSIF  i = 1 THEN
5     j := 0;
6     k := 0;
7     WHILE j < 100 DO
8        j := j + 1;
9        k := k + j
10    OD
11 FI
```

The structure of the presented program P_1 is similar to the program P_0. The second guard of the first *IF* statement has become an *ELSIF* whereas the second *IF* statement has been removed due to the applied transformation. The body of the removed *IF* statement has been copied into the modified guard of the first *IF* statement. The following Table shows the predicted set of AST types W_1 from the previous table as well as the transformation capability sets $c_c(t_1)$ and $c_p(t_1)$. Furthermore, it shows the predicted set of AST types W_2 and the set of AST types $s(P_2)$ of which the program consists. The set W_1 and W_2 have been created on the basis of the described prediction technique whereas the set $s(P_2)$ has been extracted from the program state S_2. As well as in the previous table, this set serves only to compare the precision of the prediction technique because the program state S_2 would not exist in real applications.

Table 7.2: Predicted Set of AST Types W_j of Program P_1 and P_2, Set of AST Types $c_c(t_i)$ and $c_p(t_i)$ of Transformation t_1 and Set of AST Types $s(P_i)$ of Program P_2.

W_1	$c_c(t_1)$	$c_p(t_1)$	W_2	$s(P_2)$
		T_And	T_And	
T_Assign			T_Assign	T_Assign

Continued on next page

AST types - continued from previous page.

W_1	$c_c(t_1)$	$c_p(t_1)$	W_2	$s(P_2)$
T_Assignment			T_Assignment	T_Assignment
T_Cond	T_Cond		T_Cond	T_Cond
T_Equal		T_Equal	T_Equal	T_Equal
		T_Even	T_Even	
	T_Exit		T_Exit	T_Exit
		T_False	T_False	
	T_Floop		T_Floop	T_Floop
		T_Greater	T_Greater	
		T_Greater_Eq	T_Greater_Eq	T_Greater_Eq
T_Guarded	T_Guarded		T_Guarded	T_Guarded
		T_In	T_In	
T_Less		T_Less	T_Less	
		T_Less_Eq	T_Less_Eq	
		T_Not	T_Not	
		T_Not_Eq	T_Not_Eq	
		T_Not_In	T_Not_In	
T_Number			T_Number	T_Number
		T_Odd	T_Odd	
		T_Or	T_Or	
T_Plus			T_Plus	T_Plus
T_Statements	T_Statements		T_Statements	T_Statements
T_True		T_True	T_True	
T_Variable			T_Variable	T_Variable
T_Var_Lvalue			T_Var_Lvalue	T_Var_Lvalue
T_While			T_While	

The table compares the predicted set of AST types W_j of the program P_1 with the sets of AST types $c_c(t_i)$ and $c_c(t_i)$ of the transformation t_1, with the predicted set of AST types W_j of the program P_2 and with the real set of AST types $s(P_i)$ of the program state P_2. The state P_2 is the result of applying the second transformation t_1 of the example transformation sequence on the program P_1. As shown in the table, the transformation t_1

introduces several AST types. Furthermore, some of the AST types have been removed by the transformation which has not been noticed by the prediction algorithm. This problem has previously been described.

To continue to predict whether the transformation sequence is applicable or not, the presented formula has to check the next transformation within the example sequence. For this reason, it checks if the result of the intersection of the sets W_2 and $a(t_2)$ is an empty set. This is the case because the only AST type in the set $a(t_0)$ is the specific type T_Floop which is not in the set W_2. Therefore, the transformation t_2 which is the For to While has been predicted to be inapplicable which determines that the entire transformation sequence is inapplicable. The following Listing shows the program state P_2 which is the result of applying transformation t_1 on the program P_1.

Listing 7.5: Example Program P_2 to Demonstrate the Prediction of Transformation Sequence Applicability.

```
1  IF i = 0 THEN
2    j := 10;
3    k := 20
4  ELSIF i = 1 THEN
5    j := 0;
6    k := 0;
7    DO
8      IF j >= 100 THEN
9        EXIT(1)
10     FI;
11     j := j + 1;
12     k := k + j
13   OD
14 FI
```

The Listing shows that it is impossible to apply transformation t_2 on any AST path of the program P_2 just as calculated by the prediction technique. For this fact, the entire transformation sequence can be considered as inapplicable and can be removed from a given search space.

7.4 Evaluation of Transformation Sequences

As previously described, a search space created from a transformation scheme consists of transformation sequences whose application lead to different program transformation results. In terms of constraint based program transformation theory, a search tactic attempts to find a transformation sequence which:

1. Is defined on the basis of a transformation scheme.
2. Is applicable in terms of the applicability condition of a transformation.
3. Satisfies the constraints which are included into the sequence.

The creation of the search space from a given transformation scheme as described in Section 7.2 ensures that each sequence within this space is defined by the respective scheme. Furthermore, the applicability can be predicted by a technique which has been discussed in Section 7.3. The final proof whether a sequence is applicable or not will be provided during the application of a sequence. The same applies to the proof of constraint satisfaction which can not be determined before the respective program state has been generated. The generation in turn will be achieved by applying the transformations of a respective sequence on the given initial program.

In consideration of the transformation sequence which the search tactic attempts to find, it is often beneficial to evaluate the sequences within the search space. For example, this can be achieved on the basis of transformation capabilities or effects. Afterwards, the processing order can be determined on the basis of this evaluation.

For the approach of the presented thesis, transformation effects are used to create a rating of each transformation sequence. Furthermore, a function $r_t(t_i)$ has been defined which returns the number of constraints within the sequence for which the effect of the transformation is positive. For example, the function would return 1 for the Abort Processing transformation in combination to a Number of Code Characters (NoCC) and a Cyclomatic Complexity Metric (CCM) based constraint. This is because the application of the transformation reduces the NoCC but may or may not reduce the CCM. Function $r_t(t_i)$ is described in the following where t_i is a particular transformation and $\mathbb{N}_0$ is the set of natural numbers including zero:

$r : t_i \longmapsto x \in \mathbb{N}_0$

Moreover, the following formula has been used to create a rating of an entire transformation sequence where n is the number of transformations within the sequence minus one:

$$R_S = \sum_{i=0}^{n} r(t_i)$$

The presented rating of transformation sequences can be used to evaluate the probable value of a sequence in terms of the given constraints compared to other sequences within a search space. An issue is that sequences which contain more constraints have an advantage over sequences which contain fewer constraints although the fewer constraints are possibly easier to achieve. This problem can be solved by processing transformation sequences which contain fewer constraint always first or simply divide the rating by the number of constraints within a sequence. Another issue is that different constraints are treated as equal in terms of the rating although their satisfaction might not be equally difficult. For example, the presented technique creates the same rating for a transformation sequence which contains a constraint C_1 as for another transformation sequence which contains a constraint C_2 if both sequences consist of the same transformations even if the constraint C_1 is much easier to satisfy than the constraint C_2. Moreover, the point where the constraint has to be satisfied does not affect the rating. However, the rating is often helpful and can be calculated quite fast. An application example will be provided in Chapter 9 where it is used to determine the order of transformation sequences of the same length.

7.5 Search for Transformation Paths

Chapter 6 has discussed a modelling technique which allows to outline transformation processes via so-called transformation schemes. These models define the basic conditions during searches for transformation sequences whose application on a particular program satisfy the given constraints. Furthermore, this modelling technique supports the application of transformations on unknown AST paths. For this case, the search algorithm has to check the applicability of the respective transformation on each path within the given WSL program. Since a program can consist of thousand of AST types, this is often a very time consuming process.

The situation with common transformation sequence search algorithms is even worse. Where the model based approach provides at least the possibility to manually determine the AST paths on which a transformation has to be applied, there is no such possibility with automated search algorithms. Therefore, the search space of these algorithms contains the exponential number of $(t * p_i)^n$ transformation sequences where t is the number of available transformations, p_i is the number of given AST types within the current state of the code and n is the sequence length.

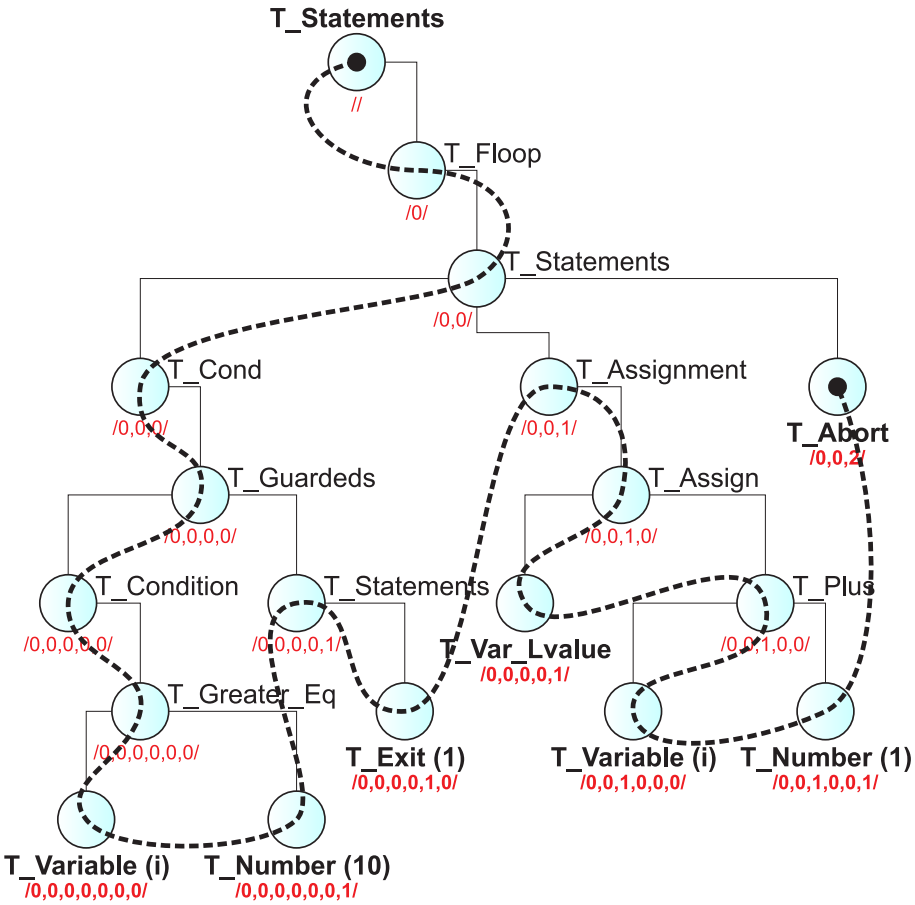
However, an approach to reduce the search space is a simple prediction technique which has been developed based on the capabilities of transformations. These capabilities have been discussed in Chapter 5. The prediction technique uses a hash map which will be created for each program state on which at least one transformation has to be applied at an abstract or unrestricted AST path. This hash map contains entries with an AST type as key and a set of AST paths as value. Moreover, it provides four basic operations which are used to handle the hash map entries:

1. **put (AST_Type key, AST_Path value)** - Puts a new entry with the referred key into the hash map and overwrites the old entry if it exists.
2. **add (AST_Type key, AST_Path value)** - Adds a value to an existing entry with the referred key. Throws an exception if the entry does not exist.
3. **boolean exists (AST_Type key)** - Returns *true* if the entry with the referred key exists or $false$ if it not exists.
4. **set < AST_Path > getPaths (AST_Type key)** - Returns the set of AST paths from the entry with the referred key. Throws an exception if the entry does not exist.

With the aid of these operations, the hash map can be filled with the AST types and paths of the respective program state. Therefore, a depth-first traversal will be applied on the AST where the path of each visited AST type will be added to the corresponding entry in the hash map. This is a complex procedure and has to be executed once per program state. The creation of the hash map and the resulting reduction of the search effort for transformation paths is important while the actual search for transformation sequences which will be discussed in the following Section. Figure 7.6 shows the traverse of an

AST with the WSL program example presented in Chapter 6.

Figure 7.6: Traverse of the AST which has been created from the WSL Program shown in Listing 6.2.



The traverse starts at the group type T_Statements which is the root of the AST. Therefore, this type has the path //. From this point, the recursive traversal algorithm propagates through the AST until it finally reaches the specific type T_Abort which has the path /0,0,2/. During this process, each of the types will be visited only once. At the visitation of a type, the algorithm either puts a new entry which contains the type and path into the hash map or it adds the current path to an existing entry. This depends on the existence of an entry with the respective type as key and is handled via the presented hash map operations. The following table presents a hash map which contains the AST types and paths that have been extracted as result of the executed algorithm.

Table 7.3: Types and Paths which have been Extracted from the WSL Program shown in Listing 6.2.

AST Type	**AST Paths**
T_Statements	{ //, /0,0/, /0,0,0,0,1/ }
T_Floop	{ /0/ }
T_Cond	{ /0,0,0/ }
T_Guardeds	{ /0,0,0,0/ }
T_Condition	{ /0,0,0,0,0/ }
T_Greater_Eq	{ /0,0,0,0,0,0/ }
T_Variable	{ /0,0,0,0,0,0,0/ }
T_Number	{ /0,0,0,0,0,0,1/, /0,0,1,0,0,1/ }
T_Exit	{ /0,0,0,0,1,0/ }
T_Assignment	{ /0,0,1/ }
T_Assign	{ /0,0,1,0/ }
T_Var_Lvalue	{ /0,0,0,0,1/ }
T_Plus	{ /0,0,1,0,0/ }
T_Variable	{ /0,0,1,0,0,0/ }
T_Abort	{ /0,0,2/ }

With the aid of this hash map and the set A of transformation capabilities, it is often possible to exclude most of the AST types on which a transformation is not applicable. It is not necessary to traverse the AST and try the applicability of a transformation each time one of the FermaT transformations has to be executed. Based on this technique, it is possible to access the paths on which a transformation is possibly applicable directly by putting each of the types within the set A as key into the hash map.

This reduces the search effort for transformations which have to be applied on an unknown AST path. In fact, the search space in these cases has been reduced to $a(t) \leq p$ on which the application of a transformation has to be checked. In this case, $a(t)$ is the number of AST types within the current state of the program which are also in the set A of the respective transformation capabilities whereas p is the number of given *AST* types within the current state of the program. This improves common automated searches for transformation sequences as well as the transformation process modelling approach

which has been discussed in Chapter 6.

For instance, the WSL example which has been traversed in Figure 7.6 consists of 18 AST types. If the transformation scheme defines that the Floop to While transformation has to be applied on an unknown AST path, the search space can be reduced from 18 to one because there is only one specific type T_Floop within the entire program on which the transformation is applicable at all.

7.6 Processing the Search Space

After the presented construction algorithm has been applied, there can be a lot of transformation sequences in the search space. The applicability prediction technique might reduce this search space but there are often many transformation sequences left. At this point, these sequences have to be searched for a particular one which is applicable in terms of the applicability condition and which satisfies the given constraints. Once this transformation has been found, the constraint based transformation process has been successfully finished and the search can be aborted.

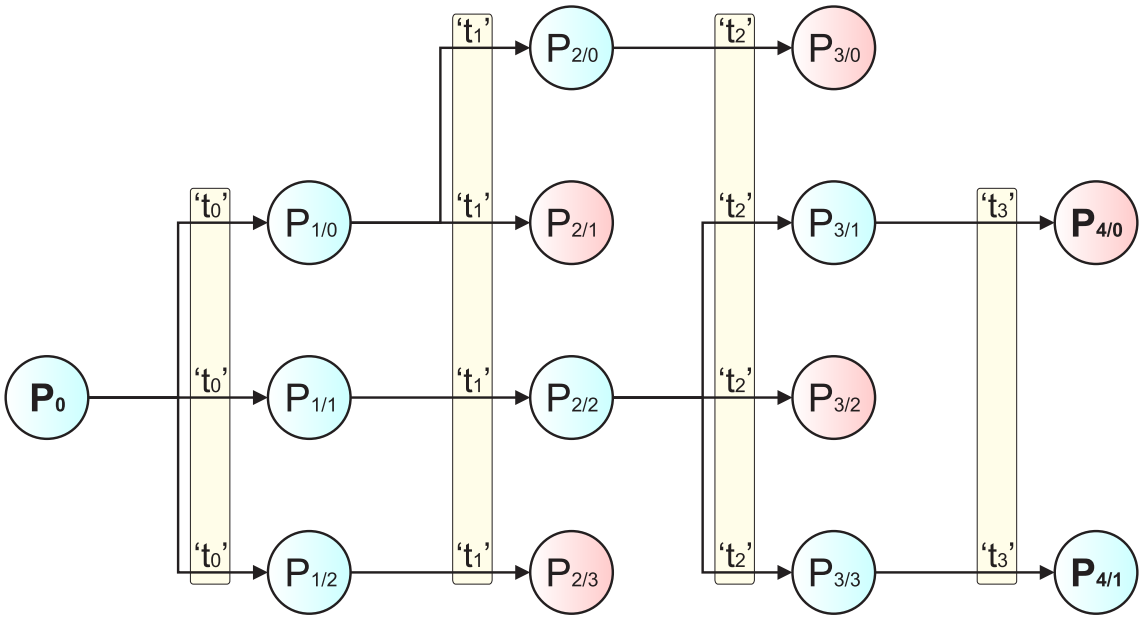
The search tactic which is used in the scope of this thesis is a simple linear search which starts at the first transformation sequence T_{S0} of the search space. To process a sequence, it has to apply each transformation within a sequence which causes the creation of a new program state after each application. Each of these states is stored as value within a hash map where the key is the sequence of transformations in combination with the definite path on which each of them has been applied to create the respective program state. If another transformation sequence will be processed, it checks if some program states can be reutilised which often reduces the computing effort.

Furthermore, the search tactic has to check if each of the constraints which are included in the sequence has been satisfied. As mentioned before, the search can be aborted if the sequence is applicable and satisfies the given constraints. If it is not applicable or it does not satisfy one of the given constraints, the next transformation sequence T_{S1} of the search space has to be processed. If all sequences within the search space have been processed without success, the constraint based transformation process is not able to provide any result and the transformation scheme which has been used to generate the search

space can be considered as overconstrained.

The application of a transformation sequence is often not that simple because some transformations have to be applied on various different AST paths within the program. This is the case if the stated path of a transformation is not definite as described in Chapter 6. It can lead to several different program versions as result of the application of a single transformation sequence. In other words, the application of a transformation sequence which contains indetermined AST paths behaves like a tree where each transformation has to be applied on its leaves. Figure 7.7 shows an example.

Figure 7.7: Application of a Transformation Sequence.



The example shows the application of a transformation sequence T_S which consists of the transformations t_0, t_1, t_2 and t_3 on an initial program P_0. The red program states are dead ends where the following transformation could not be applied or a constraint has not been satisfied whereas the program state $P_{4/1}$ is the final program.

The transformation t_0 has been applied at three different AST paths of the initial program P_0 which leads to three different program states $P_{1/0}$, $P_{1/1}$ and $P_{1/2}$. The following transformation t_1 has been applied at two different AST paths of the program state $P_{1/0}$

and at a particular AST path of the program states $P_{1/1}$ and $P_{1/2}$. This leads to four different program states $P_{2/0}$, $P_{2/1}$, $P_{2/2}$ and $P_{2/3}$. The next transformation t_2 is applicable at a particular AST path of program state $P_{2/0}$ and at three AST paths of program state $P_{2/2}$. This again leads to four different program states $P_{3/0}$, $P_{3/1}$, $P_{3/2}$ and $P_{3/3}$. The last transformation t_3 can only be applied on a particular AST path of program state $P_{3/1}$ and $P_{3/3}$. The resulting program states $P_{4/0}$ and $P_{4/1}$ of this transformation are accepted but only the state $P_{4/1}$ satisfies the given constraint. The example provides a typical application tree of a transformation sequence which contains indetermined AST paths. An AST paths in turn is indetermined if it is not definite. However, there is an application strategy for each kind of indetermined AST path:

- **Restricted AST Path** - The application strategy for restricted AST paths first checks if there already exists a hash map which contains the AST types and their corresponding AST paths. This hash map could have been previously created for transformations of other sequences which have been applied on the same program state. If it exists, the transformation uses the hash map in combination with the set of AST types A on which it is applicable to predict where it could be applied within the program. Moreover, it checks each path which has been returned from the hash map if it is valid in terms of the referred restricted AST path. This can be carried out easily because the restricted AST path can be considered as a regular expression. Afterwards, it applies the transformation at the valid AST paths. If the hash map does not exist, the hash map has to be created first which can be done directly on the stated path as described in Section 7.5. In this case, an asterisk as placeholder within the restricted AST path indicates an inclusion of the AST type at the stated path whereas a plus indicates an exclusion. A question mark in turn indicates that only the direct subtypes of the AST type at the stated path have to be considered. As mentioned before, this has been described in Chapter 6 in detail.

- **Unrestricted AST Path** - As well as the previous strategy, the application strategy for unrestricted AST paths first checks if there already exists a hash map which contains the AST types and their corresponding AST paths. If it exists, the transformation uses the hash map in combination with the set of AST types A on which it is applicable to predict where it could be applied within the program. Afterwards, it applies the transformation at these AST paths. If it does not exist, the hash map has to be created first as described in Section 7.5.

- **Abstract AST Path** - The application strategy for abstract AST paths first checks if the stated AST type is in the set of AST types *A* on which the transformation is applicable. If it is not in there, the sequence processing can be aborted. If it is in there, the strategy checks if there already exists a hash map which contains the AST types and their corresponding AST paths. If it exists, the transformation uses the hash map in combination with the stated AST type to predict where it could be applied within the program. Afterwards, it applies the transformation at these AST paths. If it does not exist, the hash map has to be created first as described in Section 7.5.

- **Abstract Restricted AST Path** - The application strategy for abstract restricted AST paths is similar to the strategy of abstract AST paths. The only difference is that it checks each path which has been returned from the hash map if it is valid in terms of the referred abstract restricted AST path. As mentioned before, this can be carried out easily because the restricted AST path can be considered as a regular expression.

7.7 Summary

The presented chapter has discussed how to deal with transformation sets. It has described the creation of the search space from a transformation scheme and it has presented a prediction technique to identify inapplicable transformation sequences. Furthermore, a technique for the evaluation of transformation sequences has been described and a particular search tactics has been discussed.

Chapter 8

Prototype Tool Support

Objectives

- To provide an overview of the Constraint Based Program Transformation System.
- To discuss the implementation of the fundamental algorithms.
- To support the research which is presented in this thesis.

The Constraint Based Program Transformation System (CBPTS) is a Java based prototype tool to support the research which is presented in this thesis. It is able to execute an entire constraint based program transformation process which includes the construction and conversion of a transformation scheme as well as the generation of the search space. The following sections discuss the implementation of the fundamental algorithms of the CBPTS.

8.1 Implementation of a Transformation Scheme

Transformation schemes have been implemented on the basis of three Java classes which are the scheme, the state and the transition. The scheme contains a set of states which are stored in a vector datastructure. Each of these states contains a set of incoming transitions

and a set of outgoing transitions. Furthermore, the scheme contains two ε-closure methods, two move method and a method which returns the set of states. The following listing shows the implementation of an ε-closure method.

Listing 8.1: Implementation of an ε-Closure Method.

```
1  public Vector<State> epsilonClosure(State state) {
2
3    /* local variables */
4    Iterator<Transition> itr;
5    Vector<State> states = new Vector<State>();
6    Transition transition;
7
8    /* add the referred state to the set of states */
9    states.add(state);
10
11   /* add the states which are reachable
12      via epsilon-transitions */
13   for (int index = 0; index < states.size(); index++) {
14     itr = states.get(index).getOutgoingTransitions()
15             .iterator();
16     while (itr.hasNext())
17       if ((transition = itr.next()).getName()
18             .equals("epsilon") &&
19           !states.contains(transition.getTarget()))
20         states.add(transition.getTarget());
21   }
22
23   return states;
24 }
```

The method contains a *while* loop surrounded by a *for* loop. The *for* loop iterates the set of states which have been reached whereas the *while* loop iterates the outgoing transitions and adds the reached states to the set of states. As mentioned before, there is another ε-closure method in the scheme class. The implementation of this class is quite similar but it takes a set of states as referred parameter rather than a single state. Apart

from the ε-closure methods, there are also two move methods provided by the scheme class. The following listing shows the implementation of one of these move methods.

Listing 8.2: Implementation of a Move Method.

```
1  public Vector<State> move(State state, String name) {
2
3    /* local variables */
4    Iterator<Transition> itr;
5    Vector<State> states = new Vector<State>();
6    Transition transition;
7
8    /* add the states which are reachable via
9       transitions with the referred name */
10   itr = state.getOutgoingTransitions().iterator();
11   while (itr.hasNext())
12     if ((transition = itr.next()).getName().equals(name) &&
13         !states.contains(transition.getTarget()))
14       states.add(transition.getTarget());
15
16   return states;
17 }
```

The method contains a *while* loop which iterates the outgoing transitions. This loop adds the states which can be reached via the referred name to the set of states.

8.2 Implementation of the Thompson Construction

As discussed in Chapter 6, a slightly adapted version of the Thompson Construction is used in the scope of this thesis. This technique has been developed by Ken Thompson in 1968 and provides a set of constructs for a stepwise creation of automata from regular expressions [79]. In particular, the slightly adapted construction technique provides the basic construct, the quantifier construct of type I, type II and type III, the alternative construct, the sequence construct and the subscheme construct. The implementation of the Thompson Construction is a Java class which contains seven methods. Each of these methods returns one of the listed constructs. The following listing shows the method

which returns a basic construct.

Listing 8.3: Implementation of the Basic Construct.

```
1  public Scheme basicConstruct(String name) {
2
3    /* local variables */
4    Scheme basicConstruct = new Scheme();
5    Vector<State> states = basicConstruct.getStates();
6
7    /* add the states to the basic construct */
8    states.add(new State());
9    states.add(new State());
10
11   /* add the transition to the basic construct */
12   states.firstElement().addTransition(
13     name,
14      states.lastElement());
15
16   /* set the initial and final state */
17   states.firstElement().setInitial(true);
18   states.lastElement().setFinal(true);
19
20   return basicConstruct;
21 }
```

The basic construct is the only constructs which is generated completely from scratch. Therefore, two new states and a new transition will be created. The only referred parameter of the method is the name of the transition which connects the first and the last state. Furthermore, the first state of the basic construct will be set initial whereas the last state will be set final. The following listing shows the method which returns a quantifier construct of type I.

Listing 8.4: Implementation of the Quantifier Construct of Type I.

```
1  public Scheme quantifierConstructTypeI(Scheme scheme) {
2
3    /* local variables */
```

```
4   Scheme quantifierConstruct = new Scheme();
5   Vector<State> states = quantifierConstruct.getStates();
6
7   /* add the states to the quantifier construct */
8   states.addAll(scheme.getStates());
9
10  /* add the transition to the quantifier construct */
11  states.firstElement().addTransition(
12    0,
13    "epsilon",
14    scheme.getStates().lastElement());
15
16  return quantifierConstruct;
17 }
```

The quantifier construct of type I takes an existing scheme and adds an ε-transition from the first state to the last state. The only referred parameter of this method is the existing scheme which will be extended. The following listing shows the method which returns a quantifier construct of type II.

Listing 8.5: Implementation of the Quantifier Construct of Type II.

```
1 public Scheme quantifierConstructTypeII(
2   Scheme scheme,
3   int quantifier) {
4
5   /* local variables */
6   Scheme quantifierConstruct = new Scheme();
7   Vector<State> states = quantifierConstruct.getStates();
8
9   /* add the states to the quantifier construct */
10  states.addAll(scheme.getStates());
11
12  /* add the transition to the quantifier construct */
13  states.lastElement().addTransition(
14    "lambda" +
```

```
15      Scheme.lambdaCounter++ +
16      "[" +
17      quantifier +
18      "]", states.firstElement());
19
20    return quantifierConstruct;
21  }
```

The quantifier construct of type II takes an existing scheme and adds a λ-transition from the last state to the first state. The referred parameters of this method are the existing scheme which will be extended and the quantifier which determines how often the lambda transition can be used. The following listing shows the method which returns a quantifier construct of type III.

Listing 8.6: Implementation of the Quantifier Construct of Type III.

```
1 public Scheme quantifierConstructTypeIII(
2   Scheme scheme,
3   int quantifier) {
4     return quantifierConstructTypeII(
5       quantifierConstructTypeI(scheme), quantifier);
6 }
```

The quantifier construct of type III is a combination of the type I and type II quantifier constructs. It takes an existing scheme and adds an ε-transition from the first state to the last state. Afterwards, it adds a λ-transition from the last state to the first state. To achieve this, it uses the other quantifier methods. The referred parameters of this method are the existing scheme which will be extended and the quantifier which determines how often the lambda transition can be used. The following listing shows the method which returns an alternative construct.

Listing 8.7: Implementation of the Alternative Construct.

```
1 public Scheme alternativeConstruct(
2   Scheme scheme_1,
3   Scheme scheme_2) {
4
5   /* local variables */
```

```
6   Scheme alternativeConstruct = new Scheme();
7   Vector<State> states = alternativeConstruct.getStates();
8
9   /* add the states to the alternative construct */
10  states.add(new State());
11  states.addAll(scheme_1.getStates());
12  states.addAll(scheme_2.getStates());
13  states.add(new State());
14
15  /* add the transitions to the alternative construct */
16  states.firstElement().addTransition(
17    "epsilon",
18    scheme_1.getStates().firstElement());
19  states.firstElement().addTransition(
20    "epsilon",
21    scheme_2.getStates().firstElement());
22  scheme_1.getStates().lastElement().addTransition(
23    "epsilon",
24    states.lastElement());
25  scheme_2.getStates().lastElement().addTransition(
26    "epsilon",
27    states.lastElement());
28
29  /* set the initial and final state */
30  states.firstElement().setInitial(true);
31  states.lastElement().setFinal(true);
32
33  /* convert the existing initial and
34     final states into common states */
35  scheme_1.getStates().firstElement().setInitial(false);
36  scheme_1.getStates().lastElement().setFinal(false);
37  scheme_2.getStates().firstElement().setInitial(false);
38  scheme_2.getStates().lastElement().setFinal(false);
39
```

```
40    return alternativeConstruct;
41  }
```

The alternative construct combines two existing transformation schemes. Therefore, a new scheme will be created. A new state will be added to this scheme followed by the states of the first existing scheme and the states of the second existing scheme. Afterwards, another new state will be added. The first new state becomes the initial state of the new scheme. It will be connected to the states of the first existing scheme and to the states of the second existing scheme via two ε-transitions. This has been discussed in Chapter 6. The second new state becomes the final state of the new scheme. It will also be connected to the states of the first existing scheme and to the states of the second existing scheme via two ε-transitions. The following listing shows the method which returns a sequence construct.

Listing 8.8: Implementation of the Sequence Construct.

```
1  public Scheme sequenceConstruct(
2    Scheme scheme_1,
3    Scheme scheme_2) {
4
5    /* local variables */
6    Scheme sequenceConstruct = new Scheme();
7    Vector<State> states = sequenceConstruct.getStates();
8
9    /* add the states to the sequence construct */
10   states.addAll(scheme_1.getStates());
11   states.addAll(scheme_2.getStates());
12
13   /* add the transition to the sequence construct */
14   scheme_1.getStates().lastElement().addTransition(
15     "epsilon",
16     scheme_2.getStates().firstElement());
17
18   /* convert some of the existing initial
19      and final states into common states */
20   scheme_1.getStates().lastElement().setFinal(false);
```

```
21    scheme_2.getStates().firstElement().setInitial(false);
22
23    return sequenceConstruct;
24  }
```

As well as the alternative construct, the sequence construct combines two existing transformation schemes. Therefore, a new scheme will be created. The states of the first existing scheme and the states of the second existing scheme will be added to this new scheme. The first state of the first scheme becomes the initial state of the new scheme whereas the last state of the second scheme becomes the final state of the new scheme. Furthermore, an ε-transition will be added to connected the states of the first existing scheme with the states of the second existing scheme. This has been discussed in Chapter 6. The following listing shows the method which returns a subscheme construct.

Listing 8.9: Implementation of the Subscheme Construct.

```
1  public Scheme subschemeConstruct(Scheme scheme) {
2
3    /* local variables */
4    Scheme subschemeConstruct = new Scheme();
5    Vector<State> states = subschemeConstruct.getStates();
6
7    /* add the states to the sequence construct */
8    states.addAll(scheme.getStates());
9    states.add(new State());
10
11   /* add the transition to the sequence construct */
12   scheme.getStates().lastElement().addTransition(
13     "epsilon",
14     states.lastElement());
15
16   /* set the final state */
17   states.lastElement().setFinal(true);
18
19   /* convert the existing final state
20      into a common state */
```

```
21    scheme.getStates().lastElement().setFinal(false);
22
23    return subschemeConstruct;
24 }
```

The subscheme construct extends an existing subscheme by a new state which will be connected to the states of the existing scheme by an ε-transition. The new state becomes the final state of the new scheme.

8.3 Implementation of the Powerset Construction

As discussed in Chapter 6, the powerset construction is based on the idea that a set of states within an ε-NFA based transformation scheme is a single state within a DFA based transformation scheme. The ε-closure methods and one of the move methods of the scheme is used to generate these sets where a state table and a transition table is created to store them. The following listing shows the implementation of the powerset construction.

Listing 8.10: Implementation of the Powerset Construction.

```
1  public Scheme construct(Scheme scheme) {
2
3    /* local variables */
4    State initialState = null;
5    Iterator<State> itr_1;
6    Iterator<Transition> itr_2;
7    Iterator<String> itr_3;
8    String name;
9    Vector<String> names = new Vector<String>();
10   Scheme newScheme = new Scheme();
11   State state;
12   Vector<State> states;
13   StateTable stateTable = new StateTable();
14   TransitionTable transitionTable = new TransitionTable();
15
16   /* prepare the scheme */
17   prepareScheme(scheme);
```

```
18
19  /* get the names of the scheme */
20  itr_1 = scheme.getStates().iterator();
21  while (itr_1.hasNext()) {
22    itr_2 = itr_1.next().getOutgoingTransitions()
23      .iterator();
24    while (itr_2.hasNext())
25      if (!(name = itr_2.next().getName())
26        .equals("epsilon"))
27        names.add(name);
28  }
29
30  /* return the scheme if there are no states */
31  if (scheme.getStates().isEmpty())
32    return newScheme;
33
34  /* get the initial state of the scheme */
35  itr_1 = scheme.getStates().iterator();
36  while (itr_1.hasNext())
37    if ((state = itr_1.next()).isInitial())
38      initialState = state;
39
40  /* apply epsilon-closure on the initial state and
41     add the resulting set to the state table */
42  stateTable.addState(scheme
43    .epsilonClosure(initialState));
44  for (int index = 0; index < stateTable
45          .getSize(); index++) {
46    itr_3 = names.iterator();
47    while (itr_3.hasNext()) {
48      states = scheme.move(
49        stateTable.getState(index),
50        (name = itr_3.next()));
51
```

```
52        if (states.isEmpty())
53          continue;
54
55        states = scheme.epsilonClosure(states);
56
57        if (!stateTable.exist(states))
58          stateTable.addState(states);
59
60        transitionTable.addTransition(
61          index,
62          stateTable.indexOf(states), name);
63      }
64
65      (state = new State()).setFinal(stateTable
66        .isFinal(index));
67      newScheme.getStates().add(state);
68    }
69
70    /* set the initial state of the new scheme */
71    newScheme.getStates().get(0).setInitial(true);
72
73    /* assemble the new scheme */
74    for (int index = 0; index < transitionTable
75           .getSize(); index++)
76      newScheme.getStates().get(transitionTable
77        .getSource(index)).addTransition(
78          transitionTable.getName(index),
79          newScheme.getStates()
80            .get(transitionTable.getTarget(index)));
81
82    return newScheme;
83  }
```

First of all, the powerset construction uses a method to prepare the scheme for the construction. This preparation process has been discussed in Chapter 6. After that, the

method uses two nested *while* loops to calculate all transition names which occur within the scheme. This calculation is followed by a condition which terminates the method and returns an empty scheme if the referred scheme contains no states. Once this condition has been passed, the initial state of the scheme will be searched. This state will be used as starting point for the powerset construction. The following *for* loop iterates the state table whereas the *while* loop which is nested inside the *for* loop iterates the transition names and adds states to the state table. The last *for* loop within this method assembles the new scheme.

8.4 Implementation of the Sequence Generator

The sequence generator is used to create all defined transformation sequences of a DFA based transformation scheme. It has to be started by calling the method with the initial state of the transformation scheme, an empty sequence and an empty hash map as parameters. Once it has been started, it traverses the scheme similar to a depth-first traversal. The following listing shows the implementation of the sequence generator.

Listing 8.11: Implementation of the Sequence Generator.

```
1  public void generate(
2    Vector<String> sequence,
3    State state,
4    HashMap<String, Integer> traverseRegistry) {
5
6    /* local variables */
7    Iterator<Transition> itr;
8    boolean stateReachable;
9    Transition transition;
10   Vector<String> newSequence;
11   HashMap<String, Integer> newTraverseRegistry;
12
13   /* add the sequence to the generated set of sequences
14      if the referred state is final */
15   if (state.isFinal())
16     addSequence(sequence);
```

```
17
18  /* process the states which are reachable
19     from the referred state */
20  itr = state.getOutgoingTransitions().iterator();
21  while (itr.hasNext()) {
22    if (hasTraverseLimit(transition = itr.next())) {
23      newTraverseRegistry = new HashMap<String, Integer>(
24        traverseRegistry);
25
26      if (stateReachable = !newTraverseRegistry
27            .containsKey(transition.getName()))
28        newTraverseRegistry.put(transition.getName(), 1);
29      else if (stateReachable = newTraverseRegistry.get(
30                  transition.getName()) <
31                  getTraverseLimit(transition))
32        newTraverseRegistry.put(
33          transition.getName(),
34          newTraverseRegistry
35            .get(transition.getName()) + 1);
36
37      if (stateReachable) {
38        newSequence = new Vector<String>();
39        newSequence.addAll(sequence);
40        generate(
41          newSequence,
42          transition.getTarget(),
43          newTraverseRegistry);
44      }
45    }
46    else {
47      newSequence = new Vector<String>();
48      newSequence.addAll(sequence);
49      newSequence.addAll(detachConstraint(transition));
50      generate(
```

```
51              newSequence,
52              transition.getTarget(),
53              traverseRegistry);
54          }
55      }
56  }
```

The implementation of the sequence generator can be split into two main parts. The first part is a condition which checks whether the current state is final or not. If it is final, a method will be called which adds the current sequence to the set of transformation sequences. This method checks if the current sequence already exist and adds it to the set in the case that it does not exist. The second part is a *while* loop which processes each outgoing transition of the current state. Therefore, a copy of the traverse history will be made for each outgoing transition. This is necessary to mark how often the critical λ-transitions have been traversed during the recursive process. Moreover, an if statement within the *while* loop checks if the current transition has a traverse limit or not. Only if it has no traverse limit, it will be added to the transformation sequence which will currently be created. This applies for λ-transitions with appended constraints as well as transformations. If the transition has a traverse limit, a new traverse registry entry will be created or the existing entry will be changed appropriately.

8.5 Summary

This chapter has presented and discussed the implementation of the fundamental algorithms of the CBPTS which is a Java based prototype tool to support the presented research. This includes algorithms for the construction and conversion of a transformation schemes as well as algorithms for the generation of the search space.

Chapter 9

Case Studies

Objectives

- To describe the use of constraint based program transformation theory on the basis of several examples.
- To demonstrate and evaluate the practical benefit of the presented approach.
- To discover and explain strengths and weaknesses of constraints and program transformation process modelling.

This chapter discusses three medium-scale case studies to describe the use of constraint based transformation theory. The first one uses a very abstract transformation scheme which includes two structure constraints to raise the abstraction level of a program. The primary intent of this case study is to prove the advantages of the proposed approach compared to search-based approaches even if the maintainer has not much knowledge about transformation theory. The second one uses a very concrete transformation scheme which includes two structure constraints to decrease the complexity of a program. The general aim of this case study is to prove the applicability of the proposed approach on larger programs. The third one uses an abstract transformation scheme which includes four behaviour constraints to decrease the execution time of a program. The main purpose of this case study is to demonstrate the assumption-based approach where constraints are used to achieve an overall target as described in Chapter 4.

9.1 Raise the Abstraction Level of a Program

Program transformation carried out with the aid of the FTE is defined as changing an initial program P_0 into a final program P_n where both programs have the same denotational semantics. However, the individual steps a program transformation process consist of are not randomly chosen. Usually, there are one or more targets which are supposed to be achieved.

The overall target in terms of this case study is to raise the abstraction level of the given program. As discussed in Chapter 4, this can be achieved by an abstraction level constraint which is satisfied if a selected group of low-level AST types do not appear within a particular program state. In this case, the selected group which is used contains only one AST type which is the T_A_S. This has been identified as low-level specific type in Table 4.2. The T_A_S is the AST type of an action system which will be described in Appendix A. An action system in turn is the fundamental basis of a low-level WSL program [74]. There must not be any actions within a program without an action system. Since the given program is based on such a system, an elimination of it during the program transformation process will be accompanied by a raise of the abstraction level.

Another target in terms of this case study is that the final program must not contain *DO* loops. This is because such loops have to contain $EXIT$ statements to terminate which usually makes them harder to comprehend than comparable FOR or $WHILE$ loops. *DO* loops are often introduced to remove recursions within an action system before it can be eliminated. The corresponding constraint to this target is a convention constraint which is satisfied if the AST type T_Floop does not appear within a particular program state. The T_Floop in turn is the specific type of a *DO* loop. The following sections will show how the development and the application of a transformation scheme which embeds these constraints can be used to model a program transformation process and to satisfy the defined constraints.

9.1.1 WSL Program Analysis

In general, a program transformation process depends a lot on the given program. For this reason, it is often beneficial to analyse the respective program at the beginning of the process. The following listing shows the WSL code of the program which is used in

the scope of this case study. It has been developed in particular to serve as program for transformation experiments.

Listing 9.1: Case Study 1: Initial Program P_0 WSL Code.

```
1  ACTIONS PROG:
2    PROG ==
3      i := 55;
4      j := k;
5      IF k < 0 THEN
6        j := j * 5;
7        CALL C
8      ELSIF k < 25 THEN
9        j := j * 2;
10       CALL C
11     FI;
12     CALL A
13   END
14   A ==
15     IF i > j THEN
16       k := k + i;
17       CALL C
18     FI;
19     CALL B
20   END
21   B ==
22     IF j < 75 THEN
23       j := j + 5;
24       k := k * 2;
25       CALL B
26     FI;
27     CALL C
28   END
29   C ==
30     i := 0;
```

```
31      CALL Z
32    END
33  ENDACTIONS
```

The program is relatively small and simple. The actual WSL code has not been created with the aid of a translator but manually written. Nevertheless, its structure looks similar to the structure of programs which have been generated through an automated translations from an assembler language. As mentioned before, the aim of the program is in particular to serve as basis for transformation experiments. Therefore, it has not been intended to be executed although it would be possible.

The given program consists of an action system which includes four actions. These actions in turn include other statements like assignments, calls or conditions. The program does not contain a *DO* loop but one of the actions is recursive. This means that a *DO* loop has to be introduced to remove the action system due to the given FermaT transformations. Afterwards, this loop has to be replaced by a loop of another type to satisfy all of the defined program transformation targets.

9.1.2 Defining the Constraints

As mentioned before, the overall target of this case study is to raise the abstraction level of the given program. Moreover, another target is that the final program P_n must not contain *DO* loops. These two targets can be considered as abstraction level constraints C_1 which defines that the specific type T_A_S has to be avoided and as convention constraint C_2 which defines that the specific type T_Floop has to be avoided. In detail, the constraints which are included in the following program transformation process are defined as follows:

1. Constraint C_1 is an abstraction level constraint which belongs to the group of high-level structure constraints. It is satisfied if the AST type T_A_S does not occur within the respective program state P_i.

2. Constraint C_2 is a convention constraint which belongs to the group of low-level structure constraints. It is satisfied if the AST type T_Floop does not occur within the respective program state P_i.

9.1.3 Transformation Scheme Development

The selection of a transformation scheme depends on the given program and the defined constraints which have to be satisfied. It is also beneficial to have a specific program transformation tactic on which the transformation scheme is based. In terms of this case study, the tactic is split into three parts. The first part is to remove recursions within the action system. The second part is to eliminate the action system which is not possible if it still contains recursive actions. The third part is to replace the *DO* loops which might have been introduced by previous transformations.

To prove the prediction technique and to simulate a maintainer with little knowledge about transformation theory, the developed transformation scheme is relatively abstract. It consists of four FermaT transformations which are the Remove Recursion in Action, the Substitute and Delete, the Simplify Item and the Floop to While. These transformations are combined via alternatives where at least one of them has to be applied and at most six of them can be applied. The number six has been determined because one recursion, three actions and one action system have to be transformed which probably requires the application of five transformations. Furthermore, the removal of the recursion will introduce a *DO* loop which has to be transformed as well. This requires the application of another transformation.

However, the Simplify Item is the only transformation within the developed transformation scheme where an AST path is stated at all. This is necessary to prevent the very versatile Simplify Item transformation from transforming statements others than the action system. As a result, the rating of the particular transformation in relation to the effect on constraint C_1 has been changed from C_1:**PX** to C_1:**P** which is shown in Table 9.1. At the end of the program transformation process, the final program P_n has to satisfy the constraints C_1 and C_2. The following listing shows the created transformation scheme described in the TSDL language.

Listing 9.2: Case Study 1: Developed Transformation Scheme Description.

```
1 (
2   (
3     < Remove Recursion in Action > |
4     < Substitute and Delete > |
```

```
5       < Simplify Item @ T_A_S > |
6       < Floop to While >
7     ) [1 .. 6]
8   ) {C1, C2}
```

The developed transformation scheme consist of an outer subscheme where the constraints C_1 and C_2 have to be satisfied after its application. This outer subscheme embeds an inner subscheme which can be applied one to six times. Furthermore, the inner subscheme contains four FermaT transformations which are combined via alternatives. The actual transformation scheme can be automatically constructed from the TSDL code as described in Chapter 6. The resulting scheme is presented in Figure 9.1.

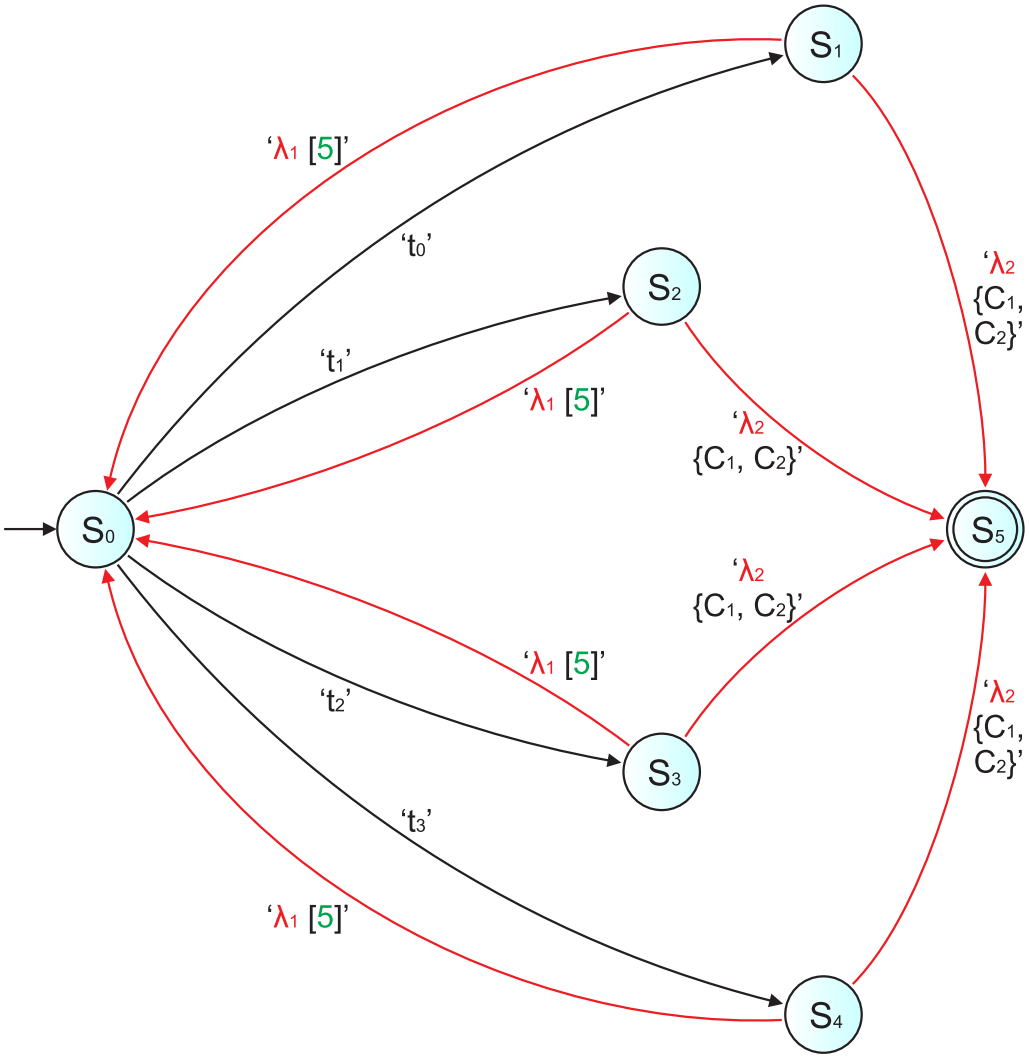
The constructed scheme is DFA based and consists of six scheme states and twelve scheme transitions of which four are transformations and eight are λ-transitions. The transformations are black coloured whereas the λ-transitions are red coloured. As described in Chapter 6, each of the λ-transitions is extended by either a constraint or a quantifier which determines how often the respective transition can be used.

The first λ-transition λ_1 is extended by a quantifier which determines that the transition can be used up to five times. In this case, the consequence is that each of the FermaT transformations can appear up to six times within a particular transformation sequence which is defined by the constructed transformation scheme. Moreover, the first λ-transition occurs four times which is a result of the powerset construction. This has been executed to create the transformation scheme. The number of these occurrences is determined by the number of transformations which have to be covered by the quantifier.

The second λ-transition λ_2 is extended by the constraints C_1 and C_2 which determine that the transition can only be used if the corresponding constraints are satisfied. Furthermore, this λ-transition is the only transition which leads to the final state S_5. Therefore, the constraints have to be satisfied in every case to achieve a successful program transformation process. As well as the first λ-transition, the second λ-transition occurs four times. The number of these occurrences is determined by the number of FermaT transformations after which the constraints have to be checked.

The constructed transformation scheme is able to generate transformation sequences

Figure 9.1: Case Study 1: Constructed Transformation Scheme.



which consist of at least one and at most six FermaT transformations. Moreover, the given constraints C_1 and C_2 appear in each transformation sequence within the search space. On the other hand, these constraints are combined to a set. Accordingly, a transformation sequence consists of at least two and at most seven elements which is discussed in Chapter 7. The following table shows the individual FermaT transformations which have been used to create the transformation scheme. Each particular transformation within this table has been rated in relation to the effects on the given constraints. As described in Chapter 5, this is required for an evaluation of transformation sequences which is an important task during the application of a transformation scheme.

Table 9.1: Case Study 1: Utilised FermaT Transformations.

ID	FermaT Transformation	Rating (C_1)	Rating (C_2)
t_0	< Remove Recursion in Action >	**X**	**N**
t_1	< Substitute and Delete @ /0,1,0/ >	**PNX**	**PNX**
t_2	< Simplify Item @ T_A_S >	**P**	**X**
t_3	< Floop to While >	**X**	**P**

The first utilised FermaT transformation t_0 has been rated as C_1:**X** and C_2:**N**. Therefore, this transformation does not increase the overall rating of a transformation sequence in which it appears. The same applies for the second utilised FermaT transformation t_1 which has been rated as C_1:**PNX** and C_2:**PNX**. The third utilised FermaT transformation t_2 is of the kind Simplify Item. This transformation is a special case because it can be used in different ways depending on the AST type on which it is applied. In terms of this case study, it is used only to delete an action system which is assured by the stated AST path. Accordingly, it has been rated as C_1:**P** and C_2:**X** which means that its appearance increases the overall rating of a transformation sequence by one. The fourth utilised FermaT transformation t_3 is no special case but it has been rated as C_1:**X** and C_2:**P**. This means that its appearance increases the overall rating of a transformation sequence by one as well.

9.1.4 Transformation Scheme Application

After the automated construction of the transformation scheme, it is possible to generate the search space which the scheme defines. This generation process is automated as well and has been described in Chapter 7. Afterwards, the search space contains 5460 transformation sequences. These are sorted by length where the shortest one will be applied first. The length in turn is defined as the number of transformations and constraints within a particular sequence.

Additionally, each transformation sequence within the search space is evaluated on the basis of the presented rating. As well as the generation of a search space, this evaluation has been described in Chapter 7. The more effects of transformations within a particular sequence are rated positive in relation to the given constraints C_1 and C_2 the higher is

the overall rate of the sequence. This overall rate in turn defines the processing order of sequences which have the same length where the highest rated one will be applied first. In cases where the length and the rate of various transformation sequences are equal, the processing order is determined by the transformation scheme.

Listing 9.3: Case Study 1: Fraction of the defined Search Space.

```
1  Sequence 1 { length = 2, rated = 1 }:
2   t2, {C1,C2}
3  Sequence 2 { length = 2, rated = 1 }:
4   t3, {C1,C2} (removed)
5  Sequence 3 { length = 2, rated = 0 }:
6   t0, {C1,C2}
7  Sequence 4 { length = 2, rated = 0 }:
8   t1, {C1,C2}
9  Sequence 5 { length = 3, rated = 2 }:
10  t2, t2, {C1,C2}
11 Sequence 6 { length = 3, rated = 2 }:
12  t2, t3, {C1,C2} (removed)
13 Sequence 7 { length = 3, rated = 2 }:
14  t3, t2, {C1,C2} (removed)
15 Sequence 8 { length = 3, rated = 2 }:
16  t3, t3, {C1,C2} (removed)
17 Sequence 9 { length = 3, rated = 1 }:
18  t0, t2, {C1,C2}
19 Sequence 10 { length = 3, rated = 1 }:
20  t0, t3, {C1,C2}
21 Sequence 11 { length = 3, rated = 1 }:
22  t1, t2, {C1,C2}
23 Sequence 12 { length = 3, rated = 1 }:
24  t1, t3, {C1,C2} (removed)
25 ...
26 Sequence 4177 { length = 7, rated = 2 }:
27  t0, t1, t1, t1, t2, t2, {C1,C2}
28 Sequence 4178 { length = 7, rated = 2 }:
```

```
 t0 , t1 , t1 , t1 , t2 , t3 , {C1,C2}
Sequence 4179 { length = 7, rated = 2 }:
 t0 , t1 , t1 , t1 , t3 , t2 , {C1,C2}
...
Sequence 5458 { length = 7, rated = 0 }:
 t1 , t1 , t1 , t1 , t0 , t1 , {C1,C2}
Sequence 5459 { length = 7, rated = 0 }:
 t1 , t1 , t1 , t1 , t1 , t0 , {C1,C2}
Sequence 5460 { length = 7, rated = 0 }:
 t1 , t1 , t1 , t1 , t1 , t1 , {C1,C2}
```

Once the search space has been created, the applicability prediction technique which has been described in Chapter 7 can be executed to identify inapplicable transformation sequences. Afterwards, these sequences can be excluded from the search space. In terms of this case study, the search space contains 5460 transformation sequences of which 2667 have been identified as inapplicable. In other words, the prediction technique was able to reduce the number of transformation sequences within the search space by 48.85%. This is a relatively high percentage which demonstrates the usefulness of this technique particularly in combination with abstract transformation schemes.

Further analysis of the prediction process has revealed that the Floop to While transformation is responsible for the massive reduction of the search space. This FermaT transformation can only be applied on the specific type T_Floop which does not appear within the initial program P_0. In contrast, the required AST types to apply the Remove Recursion in Action transformation, the Substitute and Delete transformation and the Simplify Item transformation exist within this program. Since the prediction technique does not consider the removal of AST types from a program state, these transformations are predicted to be applicable wherever they appear. This has been discussed in Chapter 7.

The described circumstances lead to the fact that each transformation sequence within the search space which contains the Floop to While transformation is inapplicable if there is no other FermaT transformation which has to be applied before and which is able to introduce the required specific type T_Floop. Furthermore, the only transformation within the constructed transformation scheme which is able to introduce this AST type is of the kind Remove Recursion in Action. In other words, a transformation sequence is applica-

ble if the Floop to While transformation is not included at all or if the Remove Recursion in Action transformation appears first within this sequence.

However, the application of the prediction technique takes only a negligible amount of time where the effort which has been made for this application is justified by the result. As mentioned before, this result is a massive reduction of the number of transformation sequences within the search space. Moreover, the determined capabilities of the used FermaT transformations on which the prediction technique is based can be reutilised in other program transformation processes.

As discussed in Chapter 7, some transformations of the search space have to be applied on various different AST paths. This can lead to a couple of different program results for the same transformation sequence. Furthermore, there are still some transformation sequences which are not applicable at all whereas other sequences are applicable but the final program P_n does not satisfy at least one of the respective constraint. Finally, the transformation sequence with the number 4178 provides the desired result. This sequence is the 1889th which has been executed because 2289 previous transformation sequences have been removed from the search space.

Unfortunately, further investigation has shown that the evaluation of the transformation sequences on the basis of transformation effects has led to a negative effect in terms of this particular case study. Without this evaluation, the transformation sequence with the number 1712 would be the first which provides the desired result. This sequence would be the 1061st which would have been executed because 651 previous transformation sequences would have been removed from the search space. The following listing shows the selected sequence and the definite AST paths on which the individual FermaT transformations have been applied.

Listing 9.4: Case Study 1: Applied Sequence of Transformations on definite AST paths to satisfy the defined Constraints.

```
1 < Remove Recursion in Action @ /0,1,2/ >,
2 < Substitute and Delete @ /0,1,1/ >,
3 < Substitute and Delete @ /0,1,1/ >,
4 < Substitute and Delete @ /0,1,1/ >,
```

```
< Simplify Item @ /0/ >,
< Floop to While @ /2,2,1,0,1,1,0/ >
```

The first FermaT transformation of the selected sequence is of the kind Remove Recursion in Action. It will be applied on the only recursive action within the action system of the initial program P_0 which has the AST path /0,1,2/. Actually, this is the only path on which the Remove Recursion in Action can be applied within the program P_0. The transformation removes the recursion and introduces a *DO* loop. On the one hand, this seems to make the program worse in terms of the given constraints. In fact, both of them are not satisfied after the application of the first transformation. On the other hand, it is necessary to remove the recursion because the Substitute and Delete transformation is not applicable on recursive actions. The following listing shows the WSL code of the program state P_1.

Listing 9.5: Case Study 1: Program State P_1 WSL Code.

```
ACTIONS PROG:
  PROG ==
    i := 55;
    j := k;
    IF k < 0 THEN
      j := j * 5;
      CALL C
    ELSIF k < 25 THEN
      j := j * 2;
      CALL C
    FI;
    CALL A
  END
  A ==
    IF i > j THEN
      k := k + i;
      CALL C
    FI;
    CALL B
  END
```

```
21   B ==
22     DO
23       IF j < 75 THEN
24         j := j + 5;
25         k := k * 2;
26         SKIP
27       ELSE
28         EXIT(1)
29       FI
30     OD;
31     CALL C
32   END
33   C ==
34     i := 0;
35     CALL Z
36   END
37 ENDACTIONS
```

The second FermaT transformation of the selected sequence is of the kind Substitute and Delete. It will be applied on the second action within the action system of the program state P_1 which has the AST path /0,1,1/. The transformation substitutes calls to the second action with the content of the second action and deletes the action afterwards. Since there is only one call to the particular action, it needs to be substituted just once. This call is included within the first action. Therefore, the substitution leads to a fusion of the first and the second action. Moreover, the action which was previously the third one within the action system becomes the second one after the application of the Substitute and Delete transformation. For this reason, it also inherits its path within the AST. The same applies to the fourth action which becomes the third within the program state P_2. The following listing shows the WSL code of the program state P_2.

Listing 9.6: Case Study 1: Program State P_2 WSL Code.

```
1 ACTIONS PROG:
2   PROG ==
3     i := 55;
4     j := k;
```

```
5       IF k < 0 THEN
6         j := j * 5;
7         CALL C
8       ELSIF k < 25 THEN
9         j := j * 2;
10        CALL C
11      FI;
12      IF i > j THEN
13        k := k + i;
14        CALL C
15      FI;
16      CALL B
17    END
18    B ==
19      DO
20        IF j < 75 THEN
21          j := j + 5;
22          k := k * 2;
23          SKIP
24        ELSE
25          EXIT(1)
26        FI
27      OD;
28      CALL C
29    END
30    C ==
31      i := 0;
32      CALL Z
33    END
34  ENDACTIONS
```

The third FermaT transformation of the selected sequence is again of the kind Substitute and Delete. It will be applied on the second action within the action system of the program state P_2 which has the AST path /0,1,1/. As mentioned before, this action has previously been the third action which was changed by the second transformation. Again, the trans-

formation substitutes calls to the second action with the content of the second action and deletes the action afterwards. Since there is again only one call to the particular action, it needs to be substituted just once. This call is included within the first action due to the fusion which has been made by the second transformation. Therefore, the substitution leads to another fusion of the first and the second action. Afterwards, the content of the first three actions within the program state P_1 is combined to the content of one action in the program state P_3. At the same time, there are only two actions left within the action system which means that the last action becomes the second one after the application of the Substitute and Delete transformation. The following listing shows the WSL code of the program state P_3.

Listing 9.7: Case Study 1: Program State P_3 WSL Code.

```
1  ACTIONS PROG:
2    PROG ==
3      i := 55;
4      j := k;
5      IF k < 0 THEN
6        j := j * 5;
7        CALL C
8      ELSIF k < 25 THEN
9        j := j * 2;
10       CALL C
11     FI;
12     IF i > j THEN
13       k := k + i;
14       CALL C
15     FI;
16     DO
17       IF j < 75 THEN
18         j := j + 5;
19         k := k * 2;
20         SKIP
21       ELSE
22         EXIT(1)
```

```
23          FI
24        OD;
25        CALL C
26      END
27      C ==
28        i := 0;
29        CALL Z
30      END
31  ENDACTIONS
```

The fourth FermaT transformation of the selected sequence is once more of the kind Substitute and Delete. It will be applied on the second action within the action system of the program state P_3 which has the AST path /0,1,1/. As mentioned before, this action has previously been the fourth action which was changed by the second and the third transformation. Once more, the transformation substitutes calls to the second action with the content of the second action and deletes the action afterwards. In contrast to the previous transformations, there are four calls to the particular action. Therefore, the content of the action needs to be substituted four times. All calls are included within the first action. In fact, this is the only action left except the one which has to be deleted by the fourth transformation. Therefore, the substitution leads to another fusion of the first and the second action. The result is that the content of all actions within the program state P_1 is combined to the content of one action in the program state P_4. At the same time, this is the only action left within the action system. The following listing shows the WSL code of the program state P_4.

Listing 9.8: Case Study 1: Program State P_4 WSL Code.

```
1  ACTIONS PROG:
2    PROG ==
3      i := 55;
4      j := k;
5      IF k < 0 THEN
6        j := j * 5;
7        i := 0;
8        CALL Z
9      ELSIF k < 25 THEN
```

```
10          j := j * 2;
11          i := 0;
12          CALL Z
13        FI;
14        IF i > j THEN
15          k := k + i;
16          i := 0;
17          CALL Z
18        FI;
19        DO
20          IF j < 75 THEN
21            j := j + 5;
22            k := k * 2;
23            SKIP
24          ELSE
25            EXIT(1)
26          FI
27        OD;
28        i := 0;
29        CALL Z
30      END
31  ENDACTIONS
```

The fifth FermaT transformation of the selected sequence is of the kind Simplify Item. It will be applied on the action system of the program state P_4 which has the AST path /0/. The transformation deletes this action system which is possible because there is only one action left within the system. After the application, only the content of this action remains. Accordingly, the resulting program state P_5 satisfies constraint C_1 because the AST type T_A_S does not occur within it. On the other hand, it does not satisfy constraint C_2 because it contains a *DO* loop. The following listing shows the WSL code of the program state P_5.

Listing 9.9: Case Study 1: Program State P_5 WSL Code.

```
1 i := 55;
2 j := k;
```

```
3  IF k < 0 THEN
4    j := j * 5;
5    i := 0
6  ELSIF k < 25 THEN
7    j := j * 2;
8    i := 0
9  ELSE
10   IF i > j THEN
11     k := k + i;
12     i := 0
13   ELSE
14     DO
15       IF j < 75 THEN
16         j := j + 5;
17         k := k * 2
18       ELSE
19         EXIT(1)
20       FI
21     OD;
22     i := 0
23   FI
24 FI
```

The sixth FermaT transformation of the selected sequence is of the kind Floop to While. It will be applied on the only *DO* loop within the program state P_5 which has the AST path /2,2,1,0,1,1,0/. The transformation converts the *DO* loop into a *WHILE* loop which appears to be a lot more tidy. After the application of the Floop to While, the resulting final program P_n satisfies not only constraint C_1 but also constraint C_2 because the AST types T_A_S and T_Floop do not occur within it. In general, the final program looks very sophisticated and could easily be translated to a high-level language like C or Java. The following listing shows the WSL code of the final program P_n.

Listing 9.10: Case Study 1: Final Program P_n WSL Code.

```
1 i := 55;
2 j := k;
```

```
3  IF k < 0 THEN
4    j := j * 5;
5    i := 0
6  ELSIF k < 25 THEN
7    j := j * 2;
8    i := 0
9  ELSE
10   IF i > j THEN
11     k := k + i;
12     i := 0
13   ELSE
14     WHILE j < 75 DO
15       j := j + 5;
16       k := k * 2
17     OD;
18     i := 0
19   FI
20 FI
```

9.2 Decrease the Complexity of a Program

The overall target in terms of this case study is to reduce the complexity of the given program. The complexity in turn has been defined by the maintainer on the basis of two software metrics which are the CCM and the NoCC. This definition depends on the assumption that a reduction of the cyclomatic complexity as well as a reduction of the program size in terms of the NoCC will improve the readability and will simplify the analysis and the comprehension of the program.

As discussed in Chapter 4, a software metric can be defined as a function which describes a particular program characteristic by means of a numeric value [24]. A metric constraint in turn can be considered as a mathematical interval which has to include this value to be satisfied. Accordingly, two specific metric constraints have to be defined in order to achieve the overall target of this case study. The following sections will show how

the development and the application of a transformation scheme which embeds these constraints can be used to model a program transformation process and to satisfy the defined constraints.

9.2.1 WSL Program Analysis

The program which is used in the scope of this case study will be presented in Appendix C. It has been developed similar to the program which has been used in the previous case study. Nevertheless, it is a lot more extensive in terms of the cyclomatic complexity and the NoCC. In fact, the program of this case study has a cyclomatic complexity of 157 and consists of 5459 code characters. It contains an action system which in turn includes 27 actions. These actions are partially very complex and include other statements like assertions, assignments, calls, conditions or loops. Even unusual statements like *WHERE* clauses or entire action systems appear within some of these 27 actions.

The actual WSL code of the program has not been created with the aid of a translator but manually written. In general, it provides a lot of possibilities for optimisations in terms of the given constraints although these optimisations are difficult to achieve. The aim of the program is in particular to serve as basis for transformation experiments. Therefore, it has not been intended to be executed although it would be possible.

9.2.2 Defining the Constraints

As mentioned before, the overall target of this case study is to reduce the complexity of the given program. The complexity in turn has been defined on the basis of the CCM and the NoCC. To achieve the overall target, these program characteristics are restricted by the constraints C_1 and C_2. In detail, the constraints which are included in the following program transformation process are defined as follows:

1. Constraint C_1 is a metric constraint which belongs to the group of high-level (structural) constraints. It is based on the CCM and is satisfied if the cyclomatic complexity of the respective program state is less than 65.

2. Constraint C_2 is a metric constraint which belongs to the group of high-level (structural) constraints. It is based on the NoCC and is satisfied if the respective program

state consists of less than 1500 code characters.

The metric constraints C_1 and C_2 have been defined on the basis of the empirical knowledge of the maintainer. Moreover, they have been chosen with the assumption that their satisfaction leads to an achievement of the overall target of the program transformation process. This case study is particularly suitable to reveal the difficulties which arise during a constraint based transformation process. In general, the definition of constraints and the effective integration of these constraints into a transformation scheme often requires a certain amount of knowledge about WSL and the FermaT transformations as well as a lot of program analysing effort.

9.2.3 Transformation Scheme Development

As discussed in the previous case study, the selection of a transformation scheme depends on the given program and the defined constraints which have to be satisfied. It is also beneficial to have a specific program transformation tactic on which the transformation scheme is based. The tactic which has been chosen for this case study is similar to the tactic which has been used in the previous case study. It is split into three parts as well but a lot more FermaT transformations are required to satisfy the defined constraints. This is because the program is a lot more extensive. Furthermore, the developed transformation scheme has to be more concrete to reduce the search space and to demonstrate how precise such a scheme can be formulated. The search space in turn has to be reduced because an application of the transformation scheme should finish in reasonable time despite the size of the given program. In other words, an application of the transformation scheme on the given program should only take minutes instead of hours or days even on a common PC which is equipped with an Intel Core 2 Duo processor and 2 GiB of main memory.

Transformation Scheme Development Part 1

The first part of the program transformation tactic tries to simplify the structure of the action system. Therefore, some FermaT transformations will be applied on the action system to delete unreachable actions and to restructure it. In terms of this case study, two transformations have been selected which are the Prune Dispatch and the Simplify Action System. The decisive factor is that the Prune Dispatch transformation will be applied first because it often removes calls to actions which possibly make some of the actions unreachable. Afterwards, these unreachable actions can be removed by the Simplify Action

System transformation.

There are a lot more FermaT transformations which could have been included like the Fix Assembler, the Fix Dispatch or the Fix Init. However, the actual WSL code of the program has not been created with the aid of a translator but manually written. This means that there are no assembler specific statements which make an application of the Fix Assembler transformation necessary. Moreover, a *WHERE* clause is undesired at this stage of the program transformation process due to the transformation tactic. This *WHERE* clause in turn will be introduced by an application of the Fix Dispatch transformation. An application of the Fix Init transformation could even remove optimisation potential because it deletes assertions which might be required to apply following transformations. This is a good example how maintainer knowledge can help to reduce the search effort. The following listing shows the created transformation scheme for this part of the program transformation tactic described in the TSDL language.

Listing 9.11: Case Study 2: Part 1 of the Transformation Scheme Description.

```
1 < Prune Dispatch @ T_Action : /0,1,0,1,?/ >,
2 < Simplify Action System @ /0,1,0/ >,
3 ...
```

The FermaT transformations which have been chosen for this part of the program transformation tactic will be applied once and in a defined order. The Prune Dispatch transformation will be applied first and on one of the actions within the action system with the AST path /0,1,0/. In fact, the specific type T_A_S at the AST path /0,1,0/ contains a group type T_Actions which has the AST path /0,1,0,1/. This group type in turn contains the general types of the actions. Accordingly, the Prune Dispatch transformation will be applied on one of these actions which is indicated by the stated path T_Action : /0,1,0,1,?/. Afterwards, the Simplify Action System will be applied on the entire action system with the AST path /0,1,0/. Since the Prune Dispatch transformation is only applicable on a dispatch action, the search tactic will try to apply the transformation on each action within the action system until it has found this particular one. Afterwards, it will carry on with the next transformation within the scheme. It is also possible to apply the Prune Dispatch on the specific type T_A_S and with it on entire action system. This would lead to the same result but the search for the dispatch action has to be carried out by the transformation itself.

Transformation Scheme Development Part 2

The second part of the program transformation tactic tries to optimise individual statements within the program. At this stage, it is very important to recognise the optimisation potential of the initial program to select beneficial FermaT transformations. For example, an application of the Abort Processing transformation is not possible if there are no *ABORT* statements within the respective program state. On the other hand, there might be some nonterminating loops or other statements within the initial program which possibly will be transformed into an *ABORT* statement during the program transformation process. This makes the prediction of suitable FermaT transformations extremely difficult. However, there are some *ABORT* and *SKIP* statements as well as assertions, conditions and even *DO*, *FOR* and *WHILE* loops within the given program. These statements are critical because they often provide optimisation potential in terms of the given constraints C_1 and C_2. Moreover, some of them indicate the application of a particular FermaT transformation.

To use the optimisation potential of the program in terms of the given constraints, four particular FermaT transformations have been selected. These are the Use Assertion transformation, the Delete All Assertions transformation, the Abort Processing transformation and the Simplify If transformation. Furthermore, three FermaT transformations which are commonly applied at this part of the program transformation tactic have been selected as well. These are the Delete All Redundant transformation, the Simplify transformation and the Constant Propagation transformation. The following listing shows the created transformation scheme for this part of the tactic described in the TSDL language.

Listing 9.12: Case Study 2: Part 2 of the Transformation Scheme Description.

```
1 ...,
2 < Use Assertion @ T_Assert > [0 .. 4],
3 < Delete All Assertions @ // >,
4 < Delete All Redundant @ // >,
5 < Abort Processing @ // >,
6 (
7   < Simplify If @ T_Cond > [0 .. 10] |
8   < Simplify @ // >
9 ) {C1},
```

```
10 < Constant Propagation @ // >,
11 ...
```

The first four of the FermaT transformations which have been chosen for this part of the program transformation tactic are combined via a sequence. Therefore, these transformations will be applied in a defined order which can be described as follows:

1. Use the assertions to simplify the program which involves the Use Assertion transformation. There are four assertions within the initial program P_0 which will be presented in Appendix C.

2. Delete the assertions and the redundant statements within the program which involves the Delete All Assertions transformation and the Delete All Redundant transformation.

3. Use the *ABORT* statements to simplify the program even more which involves the Abort Processing transformation.

The order has been determined by the maintainer. However, it is not crucial at this stage because the involved transformations affect only small parts of the program. An exception is the Delete All Assertions transformation which must not be applied before the Use Assertion transformation has been applied. The Use Assertion in turn is the only transformation within this order which is extended by a quantifier. This has been determined by the number of assertions which occur within the initial program. Therefore, this transformation will be applied up to four times at various AST paths. The other three transformations which are involved in this order will be applied just once on the root type of the AST.

The fifth and the sixth chosen transformation of the second part of the program transformation tactic are combined via an alternative. These transformations are the Simplify If and the Simplify. The intention of this alternative is that either the Simplify If transformation will be applied up to ten times on various AST paths or the Simplify transformation will be applied just once on the root type of the AST. Afterwards, the constraint C_1 has to be satisfied.

The integration of the constraint C_1 into the transformation scheme at this point has three specific reasons. In fact, these reasons are often responsible if a constraint C_j is

appended to a program state P_i which is not the final program P_n. This applies not only in terms of this case study but for almost all constraint based program transformation processes. The reasons can be described as follows:

1. The FermaT transformations which will possibly be applied after the constraint C_j has already been satisfied have been selected with the intention to satisfy another constraint C_k. In terms of this case study, these transformations have been selected to satisfy the constraint C_2 after the constraint C_1 has already been satisfied.

2. It is often beneficial to check the satisfaction of a constraints C_j as early as possible. This early check is accompanied by an early decision whether the corresponding transformation scheme T_{Si} is valid or not which often leads to a faster processing of the search space.

3. The FermaT transformations which will possibly be applied after the constraint C_j has already been satisfied have no negative effect on this constraint. In terms of this case study, an application of these transformations either leaves the cyclomatic complexity unchanged or decreases it. Accordingly, the Constant Propagation transformation and the Collapse Action System transformation have been rated as C_1:**PX** which is shown in Table 9.2.

The seventh and with it the last chosen transformation of the second part of the program transformation tactic is of the kind Constant Propagation. This transformation will be applied on the root type of the AST. However, the first four transformations, the subscheme which contains the alternative of the next two transformations and the last transformation can be considered as one sequence. As in the previous part, there could have been a lot more transformations included. In order to keep the transformation scheme as simple as possible, only a few of them which appear to be particularly suitable have been selected.

Transformation Scheme Development Part 3

The third part of the tactic finally collapses the action system which often results in comprehensible high-level code. Only a few FermaT transformations are suitable for this task which makes the selection comparatively easy. In terms of this case study, the Collapse Action System has been selected. The following listing shows the created transformation

scheme for this part of the tactic described in the TSDL language.

Listing 9.13: Case Study 2: Part 3 of the Transformation Scheme Description.

```
1 ...,
2 < Collapse Action System @ /0,1,0/ >
```

Only one FermaT transformation has been chosen for this part of the program transformation tactic. This transformation is of the kind Collapse Action System. It converts the selected action system into statements which are possibly nested inside a *DO* loop. The Collapse Action System transformation will be applied on the action system with the AST path /0,1,0/.

Completion of the Transformation Scheme

At this stage, each of the three parts of the program transformation tactic have been finished. It is now possible to combine these parts via a sequence to a single transformation scheme. Moreover, the constraint C_2 has to be satisfied at the end. Therefore, the entire scheme will be surrounded by brackets which indicate a subscheme. This subscheme in turn will be extended by the constraint C_2. It is also possible to append this constraint to the last FermaT transformation but the subscheme approach seems to be a bit easier to comprehend. The following listing shows the created transformation scheme described in the TSDL language.

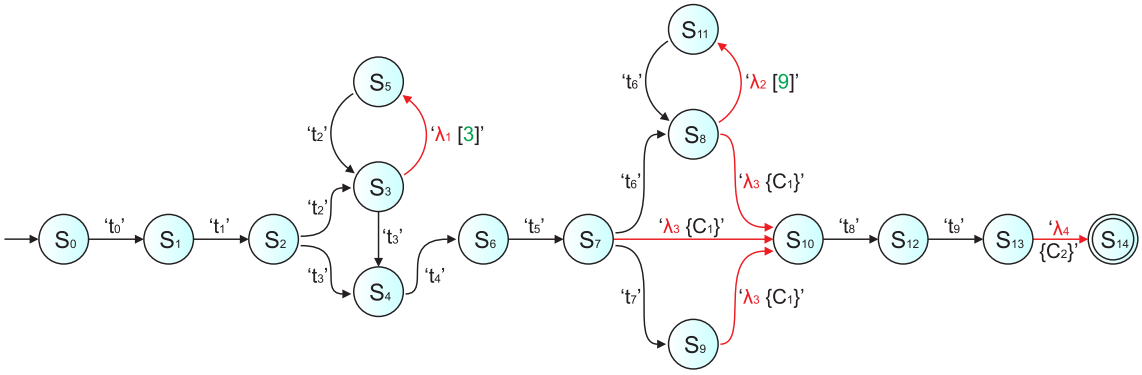
Listing 9.14: Case Study 2: Developed Transformation Scheme Description.

```
1 (
2   < Prune Dispatch @ T_Action : /0,1,0,1,?/ >,
3   < Simplify Action System @ /0,1,0/ >,
4   < Use Assertion @ T_Assert > [0 .. 4],
5   < Delete All Assertions @ // >,
6   < Delete All Redundant @ // >,
7   < Abort Processing @ // >,
8   (
9     < Simplify If @ T_Cond > [0 .. 10] |
10    < Simplify @ // >
11  ) {C1},
```

```
12    < Constant Propagation @ // >,
13    < Collapse Action System @ /0,1,0/ >
14 ) {C2}
```

As discussed before, the entire transformation scheme consists of three parts which have been developed separately. It consists of an outer subscheme where the constraint C_2 has to be satisfied after its application. This outer subscheme embeds a sequence which combines eight FermaT transformations and an inner subscheme. One of these transformations can be applied one to four times. Furthermore, the inner subscheme contains two FermaT transformations which are combined via an alternative. One of these transformations in turn can be applied one to ten times. The constraint C_1 has to be satisfied after the application of the inner subscheme. The actual transformation scheme can be automatically constructed from the TSDL code as described in Chapter 6. The resulting scheme is presented in Figure 9.2.

Figure 9.2: Case Study 2: Constructed Transformation Scheme.



The constructed scheme is DFA based and consists of 15 scheme states and 19 scheme transitions of which 13 are transformations and six are λ-transitions. The transformations are black coloured whereas the λ-transitions are red coloured. As described in Chapter 6, each of the λ-transitions is extended by either a constraint or a quantifier which determines how often the respective transition can be used.

The first λ-transition λ_1 is extended by a quantifier which determines that the transition can be used up to three times. In this case, the consequence is that the Use Assertion transformation can appear up to four times within a particular transformation sequence

which is defined by the constructed transformation scheme. The second λ-transition λ_2 is also extended by a quantifier which determines that the transition can be used up to nine times. This causes that the Simplify If transformation can appear up to ten times within a particular transformation sequence.

The third λ-transition λ_3 is extended by the constraint C_1 whereas the fourth λ-transition λ_4 is extended by the constraint C_2. Each of these transitions can only be used if the corresponding constraint is satisfied. Furthermore, there is no way to reach the final state S_{14} without using both of these λ-transitions. Therefore, the constraints have to be satisfied in every case to achieve a successful program transformation process.

The constructed transformation scheme is able to generate transformation sequences which consist of at least seven and at most 21 FermaT transformations. Moreover, the given constraints C_1 and C_2 appear in each transformation sequence within the search space. Therefore, a transformation sequence consists of at least nine and at most 23 elements which is discussed in Chapter 7. The following table shows the individual FermaT transformations which have been used to create the transformation scheme. Each particular transformation within this table has been rated in relation to the effects on the given constraints. As described in Chapter 5, this is required for an evaluation of transformation sequences which is an important task during the application of a transformation scheme.

Table 9.2: Case Study 2: Utilised FermaT Transformations.

ID	**FermaT Transformation**	**Rating** (C_1)	**Rating** (C_2)
t_0	< Prune Dispatch @ T_Action : /0,1,0,1,?/ >	**P**	**P**
t_1	< Simplify Action System @ /0,1,0/ >	**PNX**	**PNX**
t_2	< Use Assertion @ T_Assert >	**PX**	**P**
t_3	< Delete All Assertions @ // >	**X**	**P**
t_4	< Delete All Redundant @ // >	**PX**	**P**
t_5	< Abort Processing @ // >	**PX**	**P**
t_6	< Simplify If @ T_Cond >	**PX**	**P**
t_2	< Simplify @ // >	**PX**	**PNX**
t_8	< Constant Propagation @ // >	**PX**	**PNX**

Continued on next page

FermaT Transformations - continued from previous page.

ID	FermaT Transformation	Rating (C_1)	Rating (C_2)
t_9	< Collapse Action System @ /0,1,0/ >	**PX**	**PNX**

The first utilised FermaT transformation t_0 has been rated as C_1:**P** and C_2:**P** which means that its appearance increases the overall rating of a transformation sequence by two. In contrast, the second utilised FermaT transformation t_1 has been rated as C_1:**PNX** and C_2:**PNX**. Therefore, this transformation does not increase the overall rating of a transformation sequence in which it appears. An appearance of the third utilised FermaT transformation t_2 within a transformation sequence increases its overall rating by one because it has been rated as C_1:**PX** and C_2:**P**. The same applies for the fourth utilised FermaT transformation t_3 which has been rated as C_1:**X** and C_2:**P**. Moreover, the fifth, the sixth and the seventh FermaT transformation t_4, t_5 and t_6 have all been rated as C_1:**PX** and C_2:**P**. This means that an appearance of each of these transformations increases the overall rating of a transformation sequence by one as well. On the other hand, the eighth, the ninth and the tenth FermaT transformation t_7, t_8 and t_9 have all been rated as C_1:**PX** and C_2:**PNX**. For this reason, these transformations do not increase the overall rating of a transformation sequence in which they appear.

9.2.4 Transformation Scheme Application

After the automated construction of the transformation scheme, it is possible to generate the search space which the scheme defines. This generation process is automated as well and has been described in Chapter 7. Afterwards, the search space contains 60 transformation sequences. These are sorted by length where the shortest one will be applied first. The length in turn is defined as the number of transformations and constraints within a particular sequence.

Additionally, each transformation sequence within the search space is evaluated on the basis of the presented rating. As well as the generation of a search space, this evaluation has been described in Chapter 7. The more effects of transformations within a particular sequence are rated positive in relation to the given constraints C_1 and C_2 the higher is the overall rate of the sequence. This overall rate in turn defines the processing order of

sequences which have the same length where the highest rated one will be applied first. In cases where the length and the rate of various transformation sequences are equal, the processing order is determined by the transformation scheme.

Listing 9.15: Case Study 2: Fraction of the defined Search Space.

```
1  Sequence 1 { length = 9, rated = 5 }:
2   t0, t1, t3, t4, t5, {C1}, t8, t9, {C2}
3  Sequence 2 { length = 10, rated = 6 }:
4   t0, t1, t2, t3, t4, t5, {C1}, t8, t9, {C2}
5  Sequence 3 { length = 10, rated = 6 }:
6   t0, t1, t3, t4, t5, t6, {C1}, t8, t9, {C2}
7  Sequence 4 { length = 10, rated = 5 }:
8   t0, t1, t3, t4, t5, t7, {C1}, t8, t9, {C2}
9  Sequence 5 { length = 11, rated = 7 }:
10  t0, t1, t2, t2, t3, t4, t5, {C1}, t8, t9, {C2}
11 Sequence 6 { length = 11, rated = 7 }:
12  t0, t1, t2, t3, t4, t5, t6, {C1}, t8, t9, {C2}
13 Sequence 7 { length = 11, rated = 7 }:
14  t0, t1, t3, t4, t5, t6, t6, {C1}, t8, t9, {C2}
15 Sequence 8 { length = 11, rated = 6 }:
16  t0, t1, t2, t3, t4, t5, t7, {C1}, t8, t9, {C2}
17 Sequence 9 { length = 12, rated = 8 }:
18  t0, t1, t2, t2, t2, t3, t4, t5, {C1}, t8, t9, {C2}
19 Sequence 10 { length = 12, rated = 8 }:
20  t0, t1, t2, t2, t3, t4, t5, t6, {C1}, t8, t9, {C2}
21 Sequence 11 { length = 12, rated = 8 }:
22  t0, t1, t2, t3, t4, t5, t6, t6, {C1}, t8, t9, {C2}
23 Sequence 12 { length = 12, rated = 8 }:
24  t0, t1, t3, t4, t5, t6, t6, t6, {C1}, t8, t9, {C2}
25 ...
26 Sequence 36 { length = 17, rated = 13 }:
27  t0, t1, t2, t2, t2, t2, t3, t4, t5, t6, t6, t6, t6, {C1},
28  t8, t9, {C2}
29 Sequence 37 { length = 17, rated = 13 }:
```

```
30  t0 , t1 , t2 , t2 , t2 , t3 , t4 , t5 , t6 , t6 , t6 , t6 , t6 , {C1},
31  t8 , t9 , {C2}
32 Sequence 38 { length = 17, rated = 13 }:
33  t0 , t1 , t2 , t2 , t3 , t4 , t5 , t6 , t6 , t6 , t6 , t6 , t6 , {C1},
34  t8 , t9 , {C2}
35 ...
36 Sequence 58 { length = 22, rated = 18 }:
37  t0 , t1 , t2 , t2 , t2 , t2 , t3 , t4 , t5 , t6 , t6 , t6 , t6 , t6 ,
38  t6 , t6 , t6 , t6 , {C1}, t8 , t9 , {C2}
39 Sequence 59 { length = 22, rated = 18 }:
40  t0 , t1 , t2 , t2 , t2 , t3 , t4 , t5 , t6 , t6 , t6 , t6 , t6 , t6 ,
41  t6 , t6 , t6 , t6 , {C1}, t8 , t9 , {C2}
42 Sequence 60 { length = 23, rated = 19 }:
43  t0 , t1 , t2 , t2 , t2 , t2 , t3 , t4 , t5 , t6 , t6 , t6 , t6 , t6 ,
44  t6 , t6 , t6 , t6 , t6 , {C1}, t8 , t9 , {C2}
```

Once the search space has been created, the applicability prediction technique which has been described in Chapter 7 can be executed to identify inapplicable transformation sequences. Afterwards, these sequences can be excluded from the search space. Unfortunately, the prediction technique is unable to identify even one inapplicable sequence in terms of this case study. On the one hand, this indicates that the developed transformation scheme is very mature and the decisions and assumptions which have been made during the scheme development are correct. On the other hand, it also shows the limitations of this technique because there are a few transformation sequences within the search space which are inapplicable. For example, this applies for transformation sequences which contain four times the Use Assertion transformation.

However, the application of the prediction technique takes only a negligible amount of time. Afterwards, the search space still contains 60 transformation sequences because of the failure of this technique. As disussed in Chapter 7, some transformations of the search space have to be applied on various different AST paths. This can lead to different program results for the same transformation sequence. Furthermore, there are some transformation sequences which are not applicable at all whereas other sequences are applicable but at least one program state P_i does not satisfy the respective constraint. Finally, the transformation sequence with the number 37 provides the desired result.

Unfortunately, further investigation has shown that the evaluation of the transformation sequences on the basis of transformation effects has led to no effect in terms of this particular case study. Without this evaluation, the transformation sequence with the number 37 would still be the first which provides the desired result. The following listing shows the selected sequence and the definite AST paths on which the individual FermaT transformations have been applied.

Listing 9.16: Case Study 2: Applied Sequence of Transformations on definite AST paths to satisfy the defined Constraints.

```
1  < Prune Dispatch @ /0,1,0,1,26/ >,
2  < Simplify Action System @ /0,1,0/ >,
3  < Use Assertion @ /0,1,0,1,0,1,0/ >,
4  < Use Assertion @ /0,1,0,1,7,1,0/ >,
5  < Use Assertion @ /0,1,0,1,9,1,4/ >,
6  < Delete All Assertions @ // >,
7  < Delete All Redundant @ // >,
8  < Abort Processing @ // >,
9  < Simplify If @ /0,1,0,1,0,1,6,1,2,3,2/ >,
10 < Simplify If @ /0,1,0,1,1,1,0,1,1,1,0/ >,
11 < Simplify If @ /0,1,0,1,3,1,2/ >,
12 < Simplify If @ /0,1,0,1,4,1,0,1,0,3,2/ >,
13 < Simplify If @ /0,1,0,1,5,1,0/ >,
14 {C1},
15 < Constant Propagation @ // >,
16 < Collapse Action System @ /0,1,0/ >,
17 {C2}
```

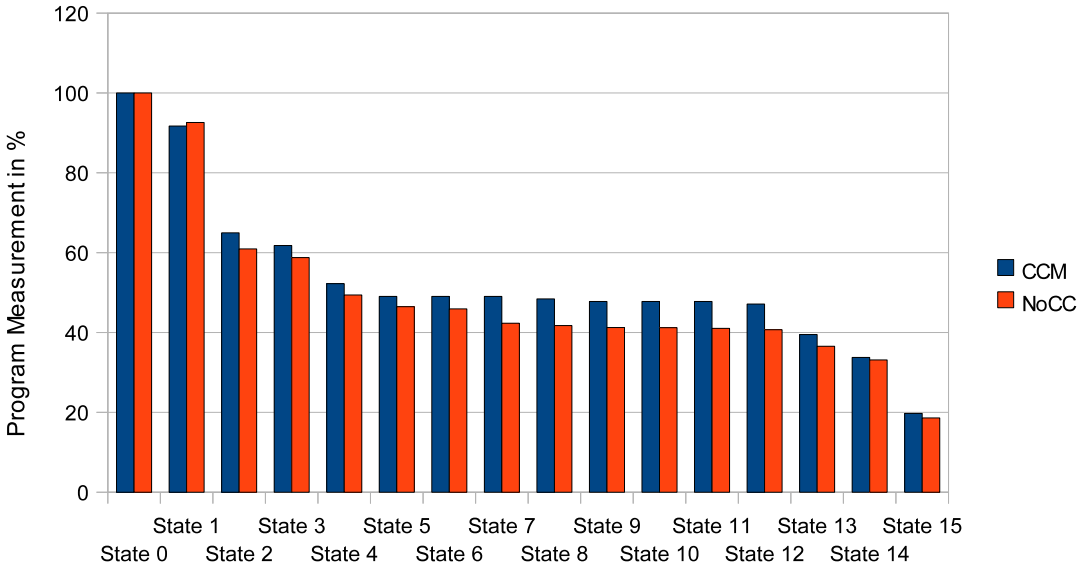
The first FermaT transformation of the selected sequence is of the kind Prune Dispatch. That was to be expected because it has been defined in the developed transformation scheme. The Prune Dispatch transformation will be applied on the only dispatch action within the action system of the initial program P_0 which has the AST path /0,1,0,1,26/. The second FermaT transformation is of the kind Simplify Action System which was to be expected as well. It will be applied on a particular AST path within the program state P_1. This path has been stated within the developed transformation scheme. The following

FermaT transformations are of the kind Use Assertion. These transformations are applied on AST paths at which assertions have been located within the respective program states P_2, P_3 and P_4. As the sequence shows, only three applications of this transformation are necessary to satisfy the given constraints C_1 and C_2 where at most four applications would have been possible in terms of the defined transformation scheme. After their application, the Delete All Assertions transformation, the Delete All Redundant transformation and the Abort Processing transformation are applied on the root type of the respective program states P_5, P_6 and P_7. This has been determined in the developed transformation scheme. The following FermaT transformations are of the kind Simplify If and are applied on AST paths where conditions have been located within the respective program states $P_8 \ldots P_{13}$. Similar to the previous application of three Use Assertion transformations, only five applications of this transformation are required to satisfy the given constraints C_1 and C_2 where at most ten applications would have been possible in terms of the defined transformation scheme. Nevertheless, the constraint C_1 is satisfied by the resulting program state P_{13} at this stage of the program transformation process and the last two FermaT transformations can be applied. These are the Constant Propagation transformation and the Collapse Action System transformation. Both FermaT transformations will be applied on particular AST paths within the program states P_{13} and P_{14}. These paths have been stated within the developed transformation scheme. Finally, the resulting final program P_n satisfies the constraint C_2. Figure 9.3 shows the evolution of the program during the application of the selected transformation sequence in relation to the defined constraint.

The Prune Dispatch transformation which has been applied on the initial program P_0 causes a larger reduction in terms of the cyclomatic complexity and the NoCC. The same applies for the Simplify Action System transformation which has been applied on the program state P_1. However, that was to be expected because these two FermaT transformations are relatively extensive. Furthermore, the Simplify Action System profits from the interplay effect of the Prune Dispatch transformation and reduces the number of actions from 27 in the program state P_1 to eleven in the program state P_2. This reduction is not only caused by the elimination of uncalled actions but also by a massive restructuring of the entire action system.

On the one hand, a comparison of the resulting program state P_2 with the previous program state P_1 shows that the Simplify Action System is indeed a complex and power-

Figure 9.3: Case Study 2: Evolution of the Transformation Sequence Application in relation to the defined Constraints.



ful FermaT transformation which can be used to simplify the entire program structure. On the other hand, it cannot be denied that some of the individual actions within the action system of the resulting program state P_2 are far more complex than the corresponding actions within the action systems of the initial program P_0 and the previous program state P_1. However, the overall measuring shows a positive development in terms of the given constraints C_1 and C_2.

The following three FermaT transformations are of the kind Use Assertion. These transformations which are applied on the respective program states P_2, P_3 and P_4 provide a larger improvement in terms of the given constraints as well. The Use Assertion transformation is able to remove entire code branches which obviously leads to success in the given case. Therefore, the cyclomatic complexity has been decreased from 102 in the program state P_2 to 77 in the program state P_5 whereas the NoCC has been decreased from 3327 in the program state P_2 to 2537 in the program state P_5. Furthermore, the three applications of this transformation do not further reduce the number of actions. However, the magnitude of the improvement in relation to the given constraints C_1 and C_2 is far bigger than expected and provides potential for further optimisations.

In contrast, the Delete All Assertions transformation has only a small impact on the given constraints C_1 and C_2. That was to be expected because this FermaT transformation only deletes assertions within a program. In terms of this case study, there are not more than four assertions within the initial program P_0. All of them have been transferred to the corresponding program state P_5 on which the Delete All Assertions transformation has been applied. After the assertions have been removed, the cyclomatic complexity has not been changed whereas the NoCC has been decreased from 2537 in the program state P_5 to 2507 in the program state P_6.

However, the impact on the cyclomatic complexity of the following FermaT transformation is more surprising. The Delete All Redundant transformation which has been applied on the program state P_6 has indeed no impact on the cyclomatic complexity. On the other hand, it decreases the NoCC from 2507 in the program state P_6 to 2310 in the program state P_7 which is a lot.

The Abort Processing transformation which has been applied on the program state P_7 decreases the cyclomatic complexity from 77 in the program state P_7 to 76 in the program state P_8 whereas it reduces the NoCC from 2310 in the program state P_7 to 2278 in the program state P_8. At this stage of the program transformation process, a further reduction of the program complexity seems to get harder. This is because the optimisation potential has largely been exhausted. Therefore, the improvement which is provided by the application of an individual transformation is not as high as at the beginning of the process. Furthermore, there are still eleven actions within the action system of the program state P_8 which means the number of actions is stable since the application of the Simplify Action System transformation.

The following five FermaT transformations are of the kind Simplify If. These transformations are very simple and there effect is confined to a fraction of the respective program states $P_8 \ldots P_{12}$ on which they have been applied. This and the largely exhausted optimisation potential are the reasons for the relatively small impact that the Simplify If transformation has on the given constraints C_1 and C_2. At least, the cyclomatic complexity has been decreased from 76 in the program state P_8 to 62 in the program state P_{13}. Accordingly, the constraint C_1 has been satisfied at the resulting program state P_{13}.

Moreover, the NoCC has been decreased from 2278 in the program state P_8 to 1996 in the program state P_{13}

The Constant Propagation transformation which has been applied on the program state P_{13} decreases the cyclomatic complexity to 53 in the resulting program state P_{14}. That was not to be expected and has no relevance because of the already satisfied constraint C_1 at the program state P_{13}. The transformation also decreases the NoCC from 1996 in the program state P_{13} to 1809 in the program state P_{14}. Afterwards, the application of the Collapse Action System transformation finally removes the action system and with it the remaining eleven actions. This is an extensive process which not only reduces the NoCC from 1809 in the program state P_{14} to 1016 in the program state P_{15}. It also reduces the cyclomatic complexity from 53 in the program state P_{14} to 36 in the program state P_{15} which was not to be expected either. However, the reduction of the NoCC leads to a satisfaction of the second constraint C_2.

At this stage of the program transformation process, the defined constraints C_1 and C_2 have been satisfied and the last FermaT transformation of the selected transformation sequence has been applied. For this reason, the program transformation process has been sucessfully finished where the current program state P_{15} is also the final program P_n. The WSL code of this program will be presented in Appendix C.

9.3 Increase the Execution Speed of a Program

The overall target in terms of this case study is to increase the execution speed of the given program. As discussed in Chapter 4, this can be achieved by execution speed constraints in combination with metric constraints. On the one hand, execution speed constraints address particular program properties which take a certain amount of time to execute. On the other hand, metric constraints determine the maximum number of execution speed constraints which are allowed to be unsatisfied.

In terms of execution speed constraints, it is assumed that program properties which take a certain amount of time to execute are slowing down the execution speed of the entire program. Accordingly, it is also assumed that a reduction of the number of these properties causes an increase of the execution speed which in turn leads to an achievement

of the overall target. However, an increased execution speed of the given program cannot be guaranteed even if all program properties which are addressed by the given execution speed constraints have been eliminated.

The overall target of the previous case study is to reduce the complexity of the given program. The complexity in turn has been defined on the basis of the CCM and the NoCC. The metric constraints which have been used in the previous case study restrict these program characteristics. Therefore, they belong to the group of structural constraints. In contrast, the metric constraints which are used in terms of this case study have been defined with the ulterior motive to increase the execution speed of the given program. Therefore, these constraints belong to the group of behavioural constraints. However, each of them is based on a software metric which can be defined as a function. This function in turn describes the non-satisfaction of execution speed constraints by means of a numeric value. Accordingly, metric constraints can be considered as mathematical intervals which have to include particular numerical values to be satisfied. The following sections will show how the development and the application of a transformation scheme which embeds these constraints can be used to model a program transformation process and to satisfy the defined constraints.

9.3.1 WSL Program Analysis

The program which is used in the scope of this case study will be presented in Appendix D. It is relatively simple and comprehensible and has been developed similar to the programs which have been used in the previous case studies. In fact, the program of this case study fills a one-dimensional array with integer values and sorts it so that the array starts with the lowest and ends with the highest value where the used algorithm is a simple bubble sort. Afterwards, the program accesses an element of the array via a binary search.

The given program consists of a main part as well as five procedures which are called within this main part. Since WSL programs cannot be compiled, the execution time of the initial program which is interpreted by FermaT 2 is $70010ms$. Once the program has been translated to C, it is possible to compile it with a common C compiler which causes an execution time decrease by more than a factor of 100. For example, the execution time of the translated program compiled with the ICC is only $483180\mu s$. The slow execution

speed of WSL programs is a result of the interpretation and limits the appropriateness of such programs in terms of time-critical comparisons. Moreover, the execution speed of C programs can be increased even more by enabling the compiler specific optimisations. This is also not possible with the version of FermaT 2 which is used in the scope of this thesis. A detailed analysis of the program execution time will be provided in Section 9.3.4.

The actual WSL code of the program has not been created with the aid of a translator but manually written. In general, it provides a lot of possibilities for optimisations in terms of the given constraints although these optimisations are difficult to achieve. The aim of the program is in particular to serve as basis for transformation experiments. In contrast to the programs which have been used in the scope of the previous case studies, it has been developed in particular to be executed.

9.3.2 Defining the Constraints

As mentioned before, the overall target in terms of this case study is to increase the execution speed of the given program. To achieve this target, the assumption-based approach of execution speed constraints is used. In this approach, execution speed constraints address particular program properties which take a certain amount of time to execute. It is assumed that this program properties are slowing down the execution speed of the entire program. Accordingly, it is also assumed that an elimination of these properties causes an increase of the execution speed which in turn leads to an achievement of the overall target. This has been discussed in Chapter 4. In detail, the execution speed constraints which are involved in the following program transformation process are defined as follows:

1. Constraint C_I is an execution speed constraint which belongs to the group of low-level (behavioural) constraints. It is based on a local program property which is the execution of an addition. The constraint is satisfied if a selected AST type within a particular program state P_i is not the specific type T_Plus.

2. Constraint C_{II} is an execution speed constraint which belongs to the group of low-level (behavioural) constraints. It is based on a local program property which is the execution of a subtraction. The constraint is satisfied if a selected AST type within a particular program state P_i is not the specific type T_Minus.

3. Constraint C_{III} is an execution speed constraint which belongs to the group of low-level (behavioural) constraints. It is based on a local program property which is the execution of a multiplication. The constraint is satisfied if a selected AST type within a particular program state P_i is not the specific type T_Times.

4. Constraint C_{IV} is an execution speed constraint which belongs to the group of low-level (behavioural) constraints. It is based on a local program property which is the execution of a division. The constraint is satisfied if a selected AST type within a particular program state P_i is not the specific type T_Div.

5. Constraint C_V is an execution speed constraint which belongs to the group of low-level (behavioural) constraints. It is based on a local program property which is the execution of an assignment. The constraint is satisfied if a selected AST type within a particular program state P_i is not the specific type T_Assign.

6. Constraint C_{VI} is an execution speed constraint which belongs to the group of low-level (behavioural) constraints. It is based on a local program property which is the execution of a procedure call. The constraint is satisfied if a selected AST type within a particular program state P_i is not the specific type T_Proc_Call.

In terms of this case study, it is assumed that an elimination of some mathematical operators, some assignments and some procedure calls within the given program will lead to an achievement of the desired overall target. The execution speed constraints C_I...C_{IV} address the mathematical operators whereas the execution speed constraint C_V addresses the assignments. The execution speed constraint C_{VI} in turn addresses the procedures calls.

As mentioned before, the defined execution speed constraints C_I...C_{VI} are used in combination with metric constraints. More precisely, metric constraints determine which program state P_i has to satisfy the defined execution speed constraints at all. However, the number of required execution speed constraints within a program state P_i depends on the number of AST types and the metric constraints which are assigned to the particular state. In fact, it has to be checked if each AST type within a program state P_i satisfies the execution speed constraints which are used by the assigned metric constraint.

This leads to the fact that the defined execution speed constraints C_I…C_{VI} will not be directly included in the transformation scheme which will be developed. As discussed in Chapter 4, these constraints serve only as basis for metric constraints. More precisely, metric constraints determine the maximum number of unsatisfied execution speed constraints within a particular program state P_i. In terms of this case study, there are four metric constraints C_1…C_4 involved in the following program transformation process. These constraints in turn are directly included in the transformation scheme which will be developed and are defined as follows:

1. Constraint C_1 is a high-level (behavioural) constraint which is based on a metric that regards the dissatisfaction of execution speed constraints. It is satisfied if the respective program state contains less than 13 dissatisfied C_I or C_{II} constraints. This leads to the fact that the constraint C_1 is satisfied if the respective program state contains less than 13 specific types which are of the kind T_Addition or T_Subtraction.

2. Constraint C_2 is a high-level (behavioural) constraint which is based on a metric that regards the dissatisfaction of execution speed constraints. It is satisfied if the respective program state contains less than four dissatisfied C_{III} or C_{IV} constraints. This leads to the fact that the constraint C_2 is satisfied if the respective program state contains less than four specific types which are of the kind T_Times or T_Div.

3. Constraint C_3 is a high-level (behavioural) constraint which is based on a metric that regards the dissatisfaction of an execution speed constraint. It is satisfied if the respective program state contains less than 25 dissatisfied C_V constraints. This leads to the fact that the constraint C_3 is satisfied if the respective program state contains less than 25 specific types which are of the kind T_Assign.

4. Constraint C_4 is a high-level (behavioural) constraint which is based on a metric that regards the dissatisfaction of an execution speed constraint. It is satisfied if the respective program state contains no dissatisfied C_{VI} constraints. This leads to the fact that the constraint C_4 is satisfied if the respective program state contains no specific types which are of the kind T_Proc_Call.

The metric constraints C_1…C_4 have been defined on the basis of the empirical knowledge of the maintainer. Moreover, they have been chosen with the assumption that their satisfaction leads to an achievement of the overall target of the program transformation

process. As mentioned before, it is difficult to prove this assumption which has to do with environmental dependencies. For example, the execution speed of the given program after the program transformation process may have increased on one but decreased on another system. In general, the definition of constraints and the effective integration of these constraints into a transformation scheme often requires a certain amount of knowledge about WSL and the FermaT transformations as well as a lot of program analysis effort.

9.3.3 Transformation Scheme Development

As discussed in the previous case studies, the selection of a transformation scheme depends on the given program and the defined constraints which have to be satisfied. It is also beneficial to have a specific program transformation tactic on which the transformation scheme is based. The tactic which has been chosen for this case study differs from the tactics which have been used in the previous case studies. This is because of two reasons. The first reason is that the given program does not contain an action system. The second reason is that the overall target in terms of this case study is to increase the execution speed of the given program. This can be considered as global property whereas the overall targets of the previous case studies can be considered as global characteristics.

However, the tactic which has been chosen for this case study is split into four parts where it is not necessary that the developed transformation scheme is as concrete as the transformation scheme which has been developed in the previous case study. This is because the given program is a lot less complex and smaller than the program which has been used in the previous case study. In fact, the program of this case study has a cyclomatic complexity of 17 and consists of 936 code characters. In general, thousands of FermaTtransformations can be applied on a program of this size in reasonable time. Reasonable time in turn means that an application of the transformation scheme only takes minutes instead of hours or days even on a common PC which is equipped with an Intel Core 2 Duo processor and 2 GiB of main memory.

Transformation Scheme Development Part 1

The first part of the program transformation tactic tries to merge consecutive assignments. This can be achieved with the aid of the Merge Left transformation or the Merge Right transformation. In terms of this case study, the Merge Right has been selected which will

be applied twice on the AST path /0,1,2,3,6,1,0/. The following listing shows a fraction of the given program which contains the affected assignments. For a better comprehension, the AST paths of the individual assignments are stated in curly brackets. These paths do of course not appear in the WSL code of the given program.

Listing 9.17: Case Study 3: Fraction of the Initial Program P_0 WSL Code.

```
1  ...;
2  low := 1;
3  high := length;
4  result := -1;
5  WHILE low <= high AND result = -1 DO
6    mid := high DIV 2;          {/0,1,2,3,6,1,0/}
7    mid := mid - low DIV 2;     {/0,1,2,3,6,1,1/}
8    mid := mid + low;           {/0,1,2,3,6,1,2/}
9    IF A[mid] > element THEN
10     high := mid - 1
11   ELSIF A[mid] < element THEN
12     low := mid + 1
13   ...
```

As a matter of fact, neither the Merge Left transformation nor the Merge Right transformation works in combination with arrays. Accordingly, the paths /0,1,2,3,6,1,0/, /0,1,2,3,6,1,1/ and /0,1,2,3,6,1,2/ are the only AST paths on which an application of one of these FermaT transformations is possible or beneficial in terms of the defined constraints. However, two applications of the Merge Right transformation are sufficient to merge the three consecutive statements. The following listing shows the created transformation scheme for this part of the program transformation tactic described in the TSDL language.

Listing 9.18: Case Study 3: Part 1 of the Transformation Scheme Description.

```
1  < Merge Right @ /0,1,2,3,6,1,0/ >,
2  < Merge Right @ /0,1,2,3,6,1,0/ >,
3  ...
```

The FermaT transformations which have been chosen for this part of the program transformation tactic will be applied twice and in a defined order. More precisely, it is a sequence which combines two Merge Right transformations. The Merge Right is a

FermaT transformation which merges the selected statement into the following statement. The selected statement in turn is in both cases the one with the AST path /0,1,2,3,6,1,0/. To put it briefly, the second Merge Right transformation will be applied on the assignment which has been created by the first application of the Merge Right transformation.

Transformation Scheme Development Part 2

The second part of the program transformation tactic tries to reduce the number of variables. This can be beneficial to satisfy the given constraints C_1, C_2 and C_3. Moreover, it tries to remove unreachable and redundant statements within the given program. This in turn can be beneficial to satisfy all of the given constraints $C_1 ... C_4$. However, four FermaT transformations which appear to be particularly suitable have been selected. These transformations are the Remove All Redundant Variables, the Constant Propagation, the Delete Unreachable Code and the Delete All Redundant. All of them will be applied on the root type of the AST. The following listing shows the created transformation scheme for this part of the program transformation tactic described in the TSDL language.

Listing 9.19: Case Study 3: Part 2 of the Transformation Scheme Description.

```
1  ...,
2  (
3    (
4      < Remove All Redundant Variables @ // > |
5      < Constant Propagation @ // > |
6      < Delete Unreachable Code @ // > |
7      < Delete All Redundant @ // >
8    ) [1 .. 4],
9    ...
10 ) {C1, C2, C3},
11 ...
```

The FermaT transformations which have been chosen for this part of the program transformation tactic are very complex. In other words, the result of each transformation is difficult to predict and their application takes a relative large amount of time. This in turn makes it difficult to use interplay effects which is one reason why an alternative combines the chosen transformations. In fact, the indetermined application order of FermaT transformations which is defined by an alternative is a lot more flexible than the

determined application order which is defined by a sequence. Therefore, an alternative supports interplay effects between various transformations a lot better than a sequence. Another reason why an alternative combines the chosen transformations is the Constant Propagation transformation. This transformation tries to simplify statements during its application which can lead to unexpected side effects. In general, these side effects are often positive in terms of the given constraints but they can be negative as well.

The alternative which combines the chosen FermaT transformations is surrounded by brackets. This indicates a subscheme which in turn is extended by a quantifier. The result is that a single transformation sequence which is defined by the developed transformation scheme can contain each of the chosen FermaT transformations. In fact, this is the reason why the number four has been determined. On the other hand, this also means that a single transformation sequence can contain a particular transformation up to four times. However, the subscheme which can be considered as inner subscheme is only the first half of another subscheme which can be labelled as outer subscheme. The second half of this outer subscheme will be developed in the following part of the program transformation tactic.

Transformation Scheme Development Part 3

The third part of the program transformation tactic tries to restructure and simplify the given program which can be beneficial to satisfy the given constraints. Therefore, two FermaT transformations which appear to be particularly suitable have been selected. These transformations are the Simplify and the Simplify Item. The Simplify transformation will be applied not more than once on the root type of the AST. On the other hand, the Simplify Item transformation will be applied up to ten times at various AST paths. The following listing shows the created transformation scheme for this part of the program transformation tactic described in the TSDL language.

Listing 9.20: Case Study 3: Part 3 of the Transformation Scheme Description.

```
1 ...,
2 (
3   ...,
4   < Simplify @ // > [0 .. 1],
5   < Simplify Item @ T_Assign > [0 .. 10]
```

```
) {C1, C2, C3},
...
```

The FermaT transformations which have been chosen for this part of the program transformation tactic are combined via a sequence. Furthermore, they are surrounded by brackets which indicate a subscheme. This subscheme in turn is the same one which is presented in Listing 9.19. Accordingly, the sequence which combines the Simplify transformation and the Simplify Item transformation is the second half of the subscheme which has been labelled as outer subscheme in the previous part of the program transformation tactic. The following listing shows a transformation scheme described in the TSDL language which combines both halves.

Listing 9.21: Case Study 3: Part 2 and 3 of the Transformation Scheme Description.

```
..., 
(
  (
    < Remove All Redundant Variables @ // > |
    < Constant Propagation @ // > |
    < Delete Unreachable Code @ // > |
    < Delete All Redundant @ // >
  ) [1 .. 4],
  < Simplify @ // > [0 .. 1],
  < Simplify Item @ T_Assign > [0 .. 10]
) {C1, C2, C3},
...
```

The FermaT transformations which have been chosen for the previous and for this part of the program transformation tactic are surrounded by brackets. These indicate the outer subscheme which in turn is extended by the constraints C_1, C_2 and C_3. The integration of these constraints into the transformation scheme at this point has three specific reasons. As discussed in the previous case study, these reasons are often responsible if a constraint C_j is appended to a program state P_i which is not the final program P_n. The reasons can be described as follows:

1. The FermaT transformations which will possibly be applied after the constraint C_j has already been satisfied have been selected with the intention to satisfy another

constraint C_k. In terms of this case study, these transformations have been selected to satisfy the constraint C_4 after the constraints C_1, C_2 and C_3 have already been satisfied.

2. It is often beneficial to check the satisfaction of a constraints C_j as early as possible. This early check is accompanied by an early decision whether the corresponding transformation scheme T_{Si} is valid or not which often leads to a faster processing of the search space.

3. The FermaT transformations which will possibly be applied after the constraint C_j has already been satisfied have no negative effect on this constraint. In terms of this case study, an application of these transformations either leaves the number of additions, subtractions, multiplications, divisions and assignments unchanged or decreases it. Accordingly, the Collapse Action System transformation has been rated as C_1:**PX** which is shown in Table 9.3.

Transformation Scheme Development Part 4

The fourth part of the tactic finally removes the procedures and with it the procedure calls which is necessary to satisfy the constraints C_4. Only a few FermaT transformations are suitable for this task which makes the selection comparatively easy. In terms of this case study, the Substitute and Delete has been selected. The following listing shows the created transformation scheme for this part of the tactic described in the TSDL language.

Listing 9.22: Case Study 3: Part 4 of the Transformation Scheme Description.

```
1 ...,
2 < Substitute and Delete @ T_Proc > [1 .. 5]
```

Only one FermaT transformation has been chosen for this part of the program transformation tactic. This transformation is of the kind Substitute and Delete. It replaces all calls to the selected procedure with its definition. Afterwards, it deletes the selected procedure. The Substitute and Delete transformation will be applied at least once and at most five times at various AST paths which is indicated by the quantifier.

Completion of the Transformation Scheme

At this stage, each of the four parts of the program transformation tactic have been finished. It is now possible to combine these parts via a sequence to a single transformation scheme. Moreover, the constraint C_4 has to be satisfied at the end. Therefore, the entire scheme will be surrounded by brackets which indicate a subscheme. This subscheme in turn will be extended by the constraint C_4. It is also possible to append this constraint to the last FermaT transformation but the subscheme approach seems to be a bit easier to comprehend. The following listing shows the created transformation scheme described in the TSDL language.

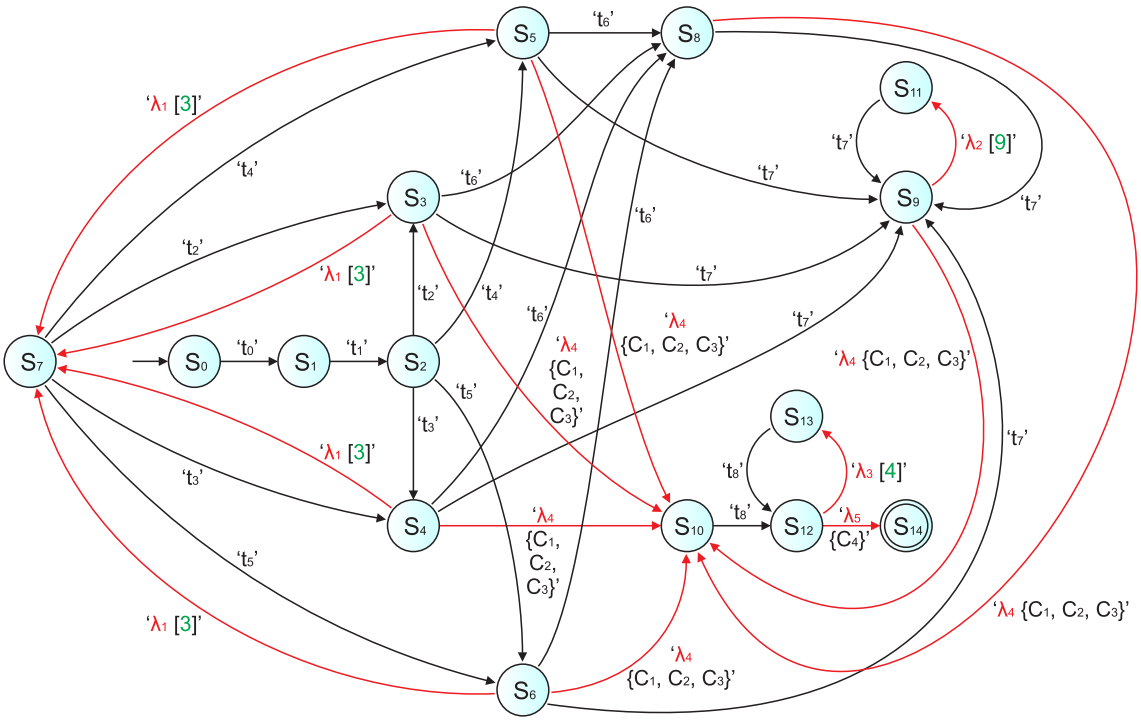
Listing 9.23: Case Study 3: Developed Transformation Scheme Description.

```
1  (
2    < Merge Right @ /0,1,2,3,6,1,0/ >,
3    < Merge Right @ /0,1,2,3,6,1,0/ >,
4    (
5      (
6        < Remove All Redundant Variables @ // > |
7        < Constant Propagation @ // > |
8        < Delete Unreachable Code @ // > |
9        < Delete All Redundant @ // >
10     ) [1 .. 4],
11     < Simplify @ // > [0 .. 1],
12     < Simplify Item @ T_Assign > [0 .. 10]
13   ) {C1, C2, C3},
14   < Substitute and Delete @ T_Proc > [1 .. 5]
15 ) {C4}
```

As discussed before, the entire transformation scheme consists of four parts which have been developed separately. It consists of an outer subscheme where the constraint C_4 has to be satisfied after its application. This outer subscheme embeds a sequence which combines three FermaT transformations and a first inner subscheme. One of these transformations will be applied one to five times. The first inner subscheme has been labelled as outer subscheme in the second and the third part of the program transformation tactic. The constraints C_1, C_2 and C_3 have to be satisfied after the application of this inner subscheme. Furthermore, it contains two FermaT transformations. One of these trans-

formations will be applied up to ten times whereas the other will be applied not more than once. The first inner subscheme also contains a second inner subscheme which in turn contains an alternative of four FermaT transformations. This second inner subscheme will be applied one to four times. The actual transformation scheme can be automatically constructed from the TSDL code as described in Chapter 6. The resulting scheme is presented in Figure 9.4.

Figure 9.4: Case Study 3: Constructed Transformation Scheme.



The constructed scheme is DFA based and consists of 15 scheme states and 35 scheme transitions of which 22 are transformations and 13 are λ-transitions. The transformations are black coloured whereas the λ-transitions are red coloured. As described in Chapter 6, each of the λ-transitions is extended by either a constraint or a quantifier which determines how often the respective transition can be used.

The first λ-transition λ_1 is extended by a quantifier which determines that the transition can be used up to three times. In this case, a consequence is that a single transformation sequence which is defined by the developed transformation scheme can contain

the Remove All Redundant Variables transformation, the Constant Propagation transformation, the Delete Unreachable Code transformation and the Delete All Redundant transformation. Another consequence is that a single transformation sequence can contain one of these FermaT transformations up to four times. The second λ-transition λ_2 is also extended by a quantifier which determines that the transition can be used up to nine times. This causes that the Simplify Item transformation can appear up to ten times within a particular transformation sequence. Another transition which is extended by a quantifier is the third λ-transition λ_3. This quantifier determines that the transition can be used up to four times which causes that the Substitute and Delete transformation can appear up to five times within a particular transformation sequence.

The fourth λ-transition λ_4 is extended by the constraints C_1, C_2 and C_3 whereas the fifth λ-transition λ_5 is extended by the constraint C_4. Each of these transitions can only be used if the corresponding constraints are satisfied. Furthermore, there is no way to reach the final state S_{14} without using both of these λ-transitions. Therefore, the constraints have to be satisfied anyway to achieve a successful program transformation process.

The constructed transformation scheme is able to generate transformation sequences which consist of at least four and at most 22 FermaT transformations. Moreover, the given constraints $C_1 \ldots C_4$ appear in each transformation sequence within the search space. On the other hand, the constraints C_1, C_2 and C_3 are combined to a set. Accordingly, a transformation sequence consists of at least six and at most 24 elements which is discussed in Chapter 7. The following table shows the individual FermaT transformations which have been used to create the transformation scheme. Each particular transformation within this table has been rated in relation to the effects on the given constraints. As described in Chapter 5, this is required for an evaluation of transformation sequences which is an important task during the application of a transformation scheme.

Table 9.3: Case Study 3: Utilised FermaT Transformations.

ID	**FermaT Transformation**	**Rating** (C_1)	**Rating** (C_2)	**Rating** (C_3)	**Rating** (C_4)
t_0	< Merge Right	**PNX**	**PNX**	**PNX**	**PNX**

Continued on next page

FermaT Transformations - continued from previous page.

ID	FermaT Transformation	Rating (C_1)	Rating (C_2)	Rating (C_3)	Rating (C_4)
	@ /0,1,2,3,6,1,0/ >				
t_1	< Merge Right @ /0,1,2,3,6,1,0/ >	**PNX**	**PNX**	**PNX**	**PNX**
t_2	< Remove All Redundant Variables @ // >	**X**	**X**	**P**	**X**
t_3	< Constant Propagation @ // >	**PX**	**PX**	**PX**	**PX**
t_4	< Delete Unreachable Code @ // >	**PX**	**PX**	**PX**	**PX**
t_5	< Delete All Redundant @ // >	**PX**	**PX**	**PX**	**PX**
t_6	< Simplify @ // >	**PX**	**PNX**	**PX**	**PX**
t_7	< Simplify Item @ T_Assign >	**PX**	**PNX**	**PX**	**PX**
t_8	< Substitute and Delete @ T_Proc >	**NX**	**NX**	**NX**	**PNX**

The first and the second utilised FermaT transformation t_0 and t_1 have been rated as C_1:**PNX**, C_2:**PNX**, C_3:**PNX** and C_4:**PNX**. Therefore, these transformations do not increase the overall rating of a transformation sequence in which they appear. In contrast, the third utilised FermaT transformation t_2 has been rated as C_1:**X**, C_2:**X**, C_3:**P** and C_4:**X** which means that its appearance increases the overall rating of a transformation sequence by one. The fourth, the fifth and the sixth FermaT transformation t_3, t_4 and t_5 have all been rated as C_1:**PX**, C_2:**PX**, C_3:**PX** and C_4:**PX**. For this reason, these transformations do not increase the overall rating of a transformation sequence in which they appear. The same applies for the seventh, the eighth and the ninth FermaT transformation t_6, t_7 and t_8. The transformations t_6 and t_7 have been rated as C_1:**PX**, C_2:**PNX**, C_3:**PX** and C_4:**PX** whereas the transformation t_8 has been rated as C_1:**NX**, C_2:**NX**, C_3:**NX** and C_4:**PNX**.

9.3.4 Transformation Scheme Application

After the automated construction of the transformation scheme, it is possible to generate the search space which the scheme defines. This generation process is automated as well and has been described in Chapter 7. Afterwards, the search space contains 37400 transformation sequences. These are sorted by length where the shortest one will be applied first. The length in turn is defined as the number of transformations and constraints within a particular sequence.

Additionally, each transformation sequence within the search space is evaluated on the basis of the presented rating. As well as the generation of a search space, this evaluation has been described in Chapter 7. The more effects of transformations within a particular sequence are rated positive in relation to the given constraints $C_1 ... C_4$ the higher is the overall rate of the sequence. This overall rate in turn defines the processing order of sequences which have the same length where the highest rated one will be applied first. In cases where the length and the rate of various transformation sequences are equal, the processing order is determined by the transformation scheme.

Listing 9.24: Case Study 3: Fraction of the defined Search Space.

```
1  Sequence 1 { length = 6, rated = 1 }:
2   t0, t1, t2, {C1,C2,C3}, t8, {C4}
3  Sequence 2 { length = 6, rated = 0 }:
4   t0, t1, t3, {C1,C2,C3}, t8, {C4}
5  Sequence 3 { length = 6, rated = 0 }:
6   t0, t1, t4, {C1,C2,C3}, t8, {C4}
7  Sequence 4 { length = 6, rated = 0 }:
8   t0, t1, t5, {C1,C2,C3}, t8, {C4}
9  Sequence 5 { length = 7, rated = 2 }:
10  t0, t1, t2, t2, {C1,C2,C3}, t8, {C4}
11 Sequence 6 { length = 7, rated = 1 }:
12  t0, t1, t2, t3, {C1,C2,C3}, t8, {C4}
13 Sequence 7 { length = 7, rated = 1 }:
14  t0, t1, t2, t4, {C1,C2,C3}, t8, {C4}
15 Sequence 8 { length = 7, rated = 1 }:
16  t0, t1, t2, t5, {C1,C2,C3}, t8, {C4}
```

```
17 Sequence 9 { length =  7, rated = 1 }:
18  t0, t1, t2, t6, {C1,C2,C3}, t8, {C4}
19 Sequence 10 { length =  7, rated = 1 }:
20  t0, t1, t2, t7, {C1,C2,C3}, t8, {C4}
21 Sequence 11 { length =  7, rated = 1 }:
22  t0, t1, t2, {C1,C2,C3}, t8, t8, {C4}
23 Sequence 12 { length =  7, rated = 1 }:
24  t0, t1, t3, t2, {C1,C2,C3}, t8, {C4}
25 ...
26 Sequence 3387 { length = 11, rated = 0 }:
27  t0, t1, t3, t5, t7, {C1,C2,C3}, t8, t8, t8, t8, {C4}
28 Sequence 3388 { length = 11, rated = 0 }:
29  t0, t1, t3, t5, {C1,C2,C3}, t8, t8, t8, t8, t8, {C4}
30 Sequence 3389 { length = 11, rated = 0 }:
31  t0, t1, t3, t6, t7, t7, t7, t7, {C1,C2,C3}, t8, {C4}
32 ...
33 Sequence 37398 { length = 24, rated = 0 }:
34  t0, t1, t5, t5, t5, t3, t6, t7, t7, t7, t7, t7, t7, t7,
35  t7, t7, t7, {C1,C2,C3}, t8, t8, t8, t8, t8, {C4}
36 Sequence 37399 { length = 24, rated = 0 }:
37  t0, t1, t5, t5, t5, t4, t6, t7, t7, t7, t7, t7, t7, t7,
38  t7, t7, t7, {C1,C2,C3}, t8, t8, t8, t8, t8, {C4}
39 Sequence 37400 { length = 24, rated = 0 }:
40  t0, t1, t5, t5, t5, t5, t6, t7, t7, t7, t7, t7, t7, t7,
41  t7, t7, t7, {C1,C2,C3}, t8, t8, t8, t8, t8, {C4}
```

As well as in the previous case study, the applicability prediction technique which has been described in Chapter 7 can be executed to identify inapplicable transformation sequences. Afterwards, these sequences can be excluded from the search space. Unfortunately, the execution result is also the same as in the previous case study. The prediction technique is unable to identify even one inapplicable sequence although the search space is a lot larger than the one of the previous case study. This shows the limitations of this technique once more. However, the developed transformation schemes of this and the previous case study have been developed with a lot of knowledge about WSL and the FermaT transformations and a lot of preceding program analysis. There are not many inapplicable

transformation sequences in the both schemes at all. This is the main reason why the presented applicability prediction technique fails. In other cases where the maintainer does not select the transformations in such an efficient way, the prediction technique has its advantages and its right to exist. For instance, this has been proven in the first case study.

However, the application of the prediction technique takes only a negligible amount of time. Afterwards, the search space still contains 37400 transformation sequences because of the failure of this technique. As discussed in Chapter 7, some transformations of the search space have to be applied on various different AST paths. This can lead to different program results for the same transformation sequence. Furthermore, there are some transformation sequences which are not applicable at all whereas other sequences are applicable but at least one program state P_i does not satisfy the respective constraint. Finally, the transformation sequence with the number 3388 provides the desired result.

Unfortunately, further investigation has shown that the evaluation of the transformation sequences on the basis of transformation effects has led to a negative effect in terms of this particular case study. Without this evaluation, the transformation sequence with the number 2902 would be the first which provides the desired result. In general, it seems that the sequence evaluation does not work as expected because it often decelerates the application of a transformation scheme. In contrast, the model based prediction technique is not able to appreciably decelerate the application of a scheme. On the other hand, it does not accelerate the application of well defined transformation schemes either. The following listing shows the selected sequence and the definite AST paths on which the individual FermaT transformations have been applied.

Listing 9.25: Case Study 3: Applied Sequence of Transformations on definite AST paths to satisfy the defined Constraints.

```
1 < Merge Right @ /0,1,2,3,6,1,0/ >,
2 < Merge Right @ /0,1,2,3,6,1,0/ >,
3 < Constant Propagation @ // >,
4 < Delete All Redundant @ // >,
5 {C1, C2, C3},
6 < Substitute and Delete @ /0,1,0/ >,
7 < Substitute and Delete @ /0,1,0/ >,
```

```
8  < Substitute and Delete @ /0,1,0/ >,
9  < Substitute and Delete @ /0,1,0/ >,
10 < Substitute and Delete @ /0,1,0/ >,
11 {C4}
```

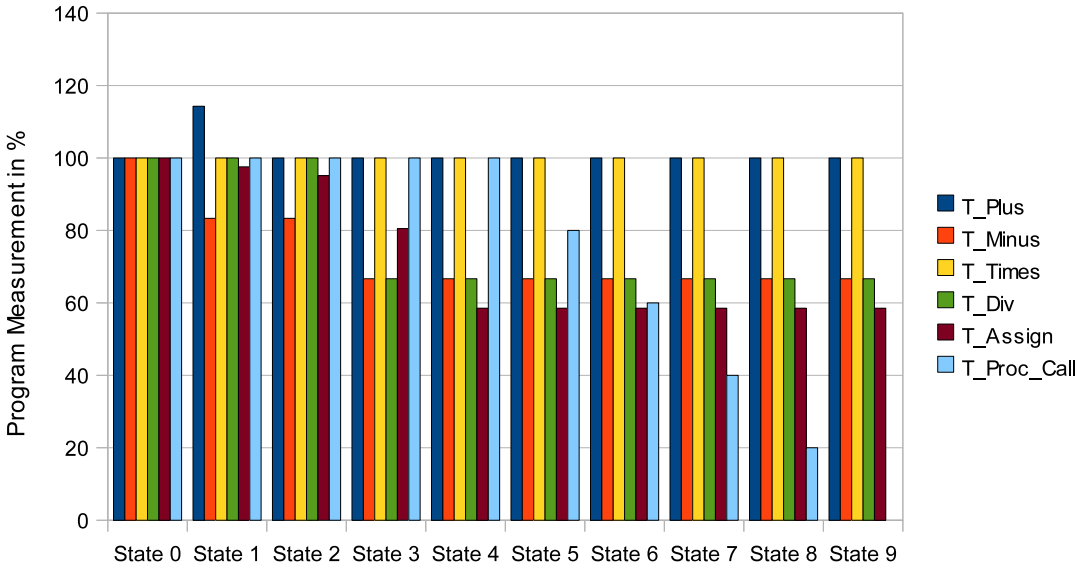
The first FermaT transformation of the selected sequence is of the kind Merge Right. That was to be expected because it has been defined in the developed transformation scheme. The transformation will be applied on the AST path /0,1,2,3,6,1,0/ of the initial program P_0 to eliminate a particular assignment. The second FermaT transformation is also of the kind Merge Right. This transformation will be applied on the AST path /0,1,2,3,6,1,0/ of the respective program state P_1. It eliminates another assignment and was defined in the developed transformation scheme as well. The third FermaT transformation is of the kind Constant Propagation whereas the fourth FermaT transformation is of the kind Delete All Redundant. These transformations are part of the alternative which has been developed in the second part of the program transformation tactic. They will be applied on the root type of the respective program states P_2 and P_3. However, the resulting program state P_4 satisfies the constraints C_1, C_2 and C_3. According to the transformation scheme, the following five FermaT transformations are of the kind Substitute and Delete. These transformations are applied on the AST path /0,1,0/ of the respective program states $P_4 \ldots P_8$. Finally, the resulting program state P_9 satisfies the last remaining constraint C_4.

A lot of FermaT transformations which appear in the developed transformation scheme do not appear within the selected transformation sequence. This is often the case with relative complex transformation schemes. It indicates that the scheme might be a little underconstrained. Figure 9.5 shows the evolution of the program during the application of the selected transformation sequence in relation to the defined constraints.

The Merge Right transformation which has been applied on the initial program P_0 creates one addition. On the other hand, this FermaT transformation eliminates one subtraction as well as one assignment. The number of other mathematical operators and the number of procedure calls remain unchanged. Therefore, there are eight additions, five subtractions, one multiplication, three divisions, 40 assignments and five procedure calls within the program state P_1 which was to be expected.

The following FermaT transformation is of the kind Merge Right as well. This trans-

Figure 9.5: Case Study 3: Evolution of the Transformation Sequence Application in relation to the defined Constraints.



formation has been applied on the program state P_1 and eliminates one addition and one assignment. The number of other mathematical operators and the number of procedure calls remain unchanged. Therefore, there are seven additions, five subtractions, one multiplication, three divisions, 39 assignments and five procedure calls within the program state P_2 which was to be expected as well.

The Constant Propagation transformation has been applied on the program state P_2. This FermaT transformation eliminates one subtraction, one division as well as six assignments. The number of other mathematical operators and the number of procedure calls remain unchanged. Therefore, there are seven additions, four subtractions, one multiplication, two divisions, 33 assignments and five procedure calls within the program state P_3. Especially the reduction of the assignments is a massive improvement which was not to be expected. Therefore, the effect of the Constant Propagation transformation in terms of the constraint C_3 is a reason why the developed transformation scheme seems to be a little underconstrained.

The Delete All Redundant transformation which has been applied on the program state

P_3 eliminates nine assignments. The number of mathematical operators and the number of procedure calls remain unchanged. Therefore, there are seven additions, four subtractions, one multiplication, two divisions, 24 assignments and five procedure calls within the program state P_4. The massive reduction of the assignments was not to be expected either and leads to a satisfaction of the defined constraints C_1, C_2 and C_3 at this stage of the program transformation process. Accordingly, the effect of the Delete All Redundant transformation in terms of the constraint C_3 is another reason why the developed transformation scheme seems to be a little underconstrained.

The following five transformations are of the kind Substitute and Delete. These FermaT transformations have been applied on the first procedure within the respective program states $P_4 \ldots P_8$. The result is a linear reduction of the number of procedures from five in the program state P_4 to zero in the program state P_9. Once all procedures have been removed, there are also no procedure calls left and the last defined constraint C_4 has been satisfied as well. The number of mathematical operators and the number of assignments remain unchanged. Therefore, there are seven additions, four subtractions, one multiplication, two divisions, 24 assignments and zero procedure calls within the program state P_9.

At this stage of the program transformation process, the defined constraints $C_1 \ldots C_4$ have been satisfied and the last FermaT transformation of the selected transformation sequence has been applied. For this reason, the program transformation process has been successfully finished where the current program state P_9 is also the final program P_n. The WSL code of this program will be presented in Appendix D.

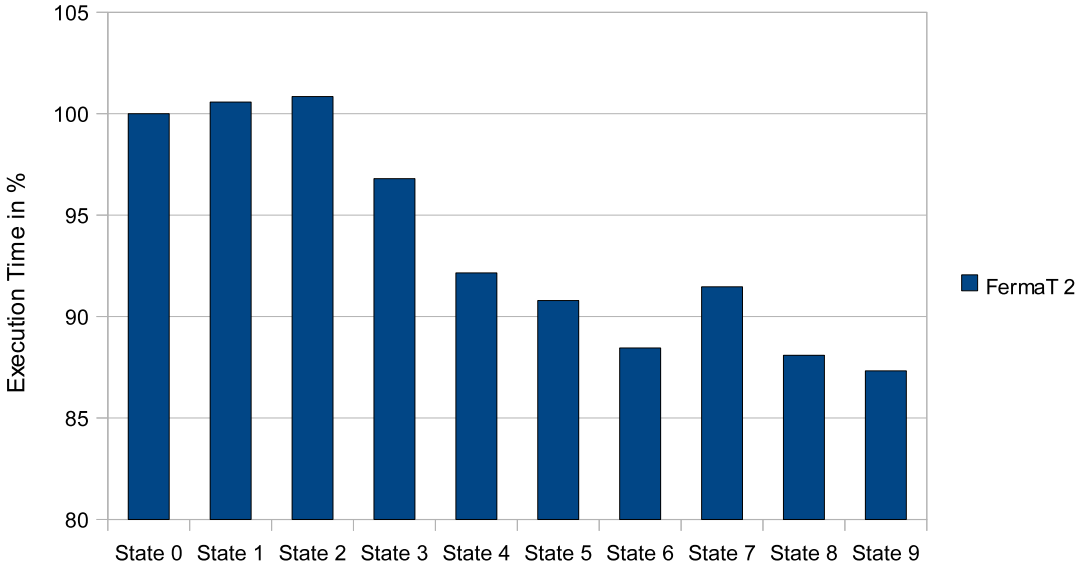
9.3.5 Execution Speed Comparison of the Original Program States

As mentioned before, the overall target in terms of this case study is to increase the execution speed of the given program. To achieve this, an assumption-based approach has been chosen which uses the execution speed constraints $C_I \ldots C_{VI}$ in combination with the metric constraints $C_1 \ldots C_4$. On the one hand, the execution speed constraints address particular program properties which take a certain amount of time to execute. On the other hand, the metric constraints determine the maximum number of execution speed constraints which are allowed to be unsatisfied within a particular program state P_i. In terms of execution

speed constraints, it is assumed that program properties which take a certain amount of time to execute are slowing down the execution speed of the entire program. Accordingly, it is also assumed that a reduction of the number of these properties causes an increase of the execution speed which in turn leads to an achievement of the overall target.

However, a lot of execution speed comparisons have been carried out to prove whether these assumptions are correct or not. As described in Chapter 4, these comparisons depend a lot on the given environment and the circumstances which have been present. In this case, each of the individual measurements has been taken on a PC equipped with an Intel Core 2 Duo processor, 2 GiB of main memory and Windows XP Professional SP2. The results are probably different in combination with other environments but a tendency is recognisable. Figure 9.6 shows the execution time of the initial program P_0, the final program P_n and all other program states $P_1 \ldots P_8$.

Figure 9.6: Case Study 3: Comparison of the Program States $P_0 \ldots P_9$ interpreted by FermaT 2.



The program states $P_0 \ldots P_9$ have not been translated to another language for this execution time comparison. In fact, the WSL code of these states was interpreted by FermaT 2 which in turn uses an installation of Active Perl v5.8.9.826. As mentioned before, this

causes a very slow execution. Nevertheless, the comparison is significant because all of the included measurements have been taken under the same conditions.

The application of the first FermaT transformation increases the execution time from 70010*ms* of the program state P_0 to 70412*ms* of the program state P_1. That was not to be expected because the program state P_1 contains less assignments than the initial program P_0 whereas the number of mathematical operators and procedure calls is the same. The application of the second FermaT transformation also increases the execution time from 70412*ms* of the program state P_1 to 70604*ms* of the program state P_2. That was not to be expected either because the program state P_2 contains not only less assignments but also less mathematical operators than the program state P_1 whereas the number of procedure calls is the same. In contrast, the third FermaT transformation decreases the execution time from 70604*ms* of the program state P_2 to 67771*ms* of the program state P_3. The same applies for the fourth and the fifth FermaT transformation. More precisely, the fourth transformation decreases the execution time from 67771*ms* of the program state P_3 to 64516*ms* of the program state P_4 whereas the fifth transformation decreases the execution time from 64516*ms* of the program state P_4 to 63562*ms* of the program state P_5. The remaining five FermaT transformations eliminate the procedures within the program. The average effect of this transformations is positive but the elimination of the third procedure by the seventh transformation increases the execution time from 61927*ms* of the program state P_6 to 64036*ms* of the program state P_7. It was not possible to find a particular reason for this increase but it seems to depend on the used environment. However, the execution time of the final program P_n is only 61135*ms* whereas the execution time of the initial program P_0 is 70010*ms*. In other words, the execution speed of the program has been increased by 14.52% in this particular case.

9.3.6 Execution Speed Comparison of the Translated Program States

In general, it is required to translate the WSL code of a program to another language for commercial use. A main reason for this is that WSL code cannot be compiled and that the interpretation of such code is quite slow. To get a more practice-oriented comparison, the initial program P_0, the final program P_n and all other program states $P_1 \ldots P_8$ have been translated to C. Afterwards, these states have been compiled with four different C compilers which are:

1. the GNU C Compiler (GCC) v2.95.2,
2. the GNU C Compiler (GCC) v4.4.0,
3. the Intel C Compiler (ICC) v11.0.072 and
4. the Microsoft C Compiler (MCC) v9.0.21022.8.

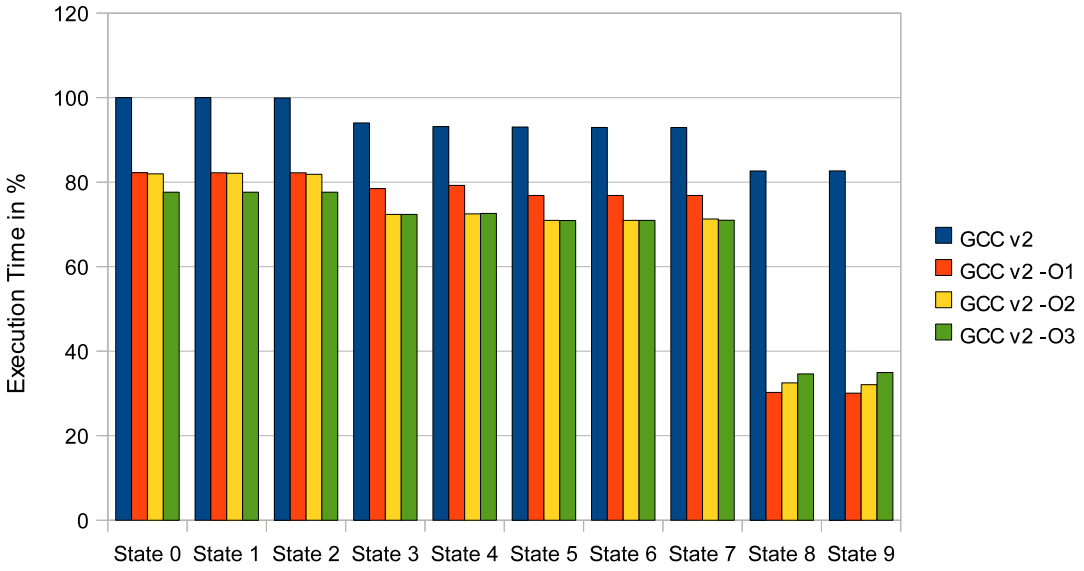
The execution time measurements of the translated program states have been taken on a PC which was equipped with an Intel Core 2 Duo processor, 2 GiB of main memory and Windows XP Professional SP2. These measurements will be discussed and compared in the following sections.

Gnu C Compiler v2

The translated program states $P_0 \ldots P_9$ of this execution time comparison have been compiled with the GCC v2. This compiler provides three general optimisation options in terms of execution speed. Accordingly, there exist four versions of each translated state where the first of these versions has been compiled without any optimisation at all. The other three have been compiled with the optimisation option -O1, -O2 and -O3. Figure 9.7 shows the execution time of the translated states.

The application of the first FermaT transformation has almost no effect on the execution speed of the program. In other words, the program state P_0 and the program state P_1 are almost equally fast. The same applies to the application of the second FermaT transformation which means that the program state P_1 and the program state P_2 are almost equally fast as well. This partially confirms the result of the previous comparison where the original program states have been interpreted by FermaT 2. The application of the third FermaT transformation decreases the execution time from 502030μs, 413030μs (-O1), 411230μs (-O2) and 389930μs (-O3) of the program state P_2 to 472300μs, 394230μs (-O1), 363600μs (-O2) and 363600μs (-O3) of the program state P_3. This confirms the result of the previous comparison. The application of the following four FermaT transformations in turn has almost no effect on the execution speed of the program. This means that the program state P_3, the program state P_4, the program state P_5, the program state P_6 and the program state P_7 are almost equally fast. That was not to be expected especially in regard to the fourth transformation which decreases the number of assignments from 33 in the program state P_3 to 24 in the program state P_4. Furthermore, it does not

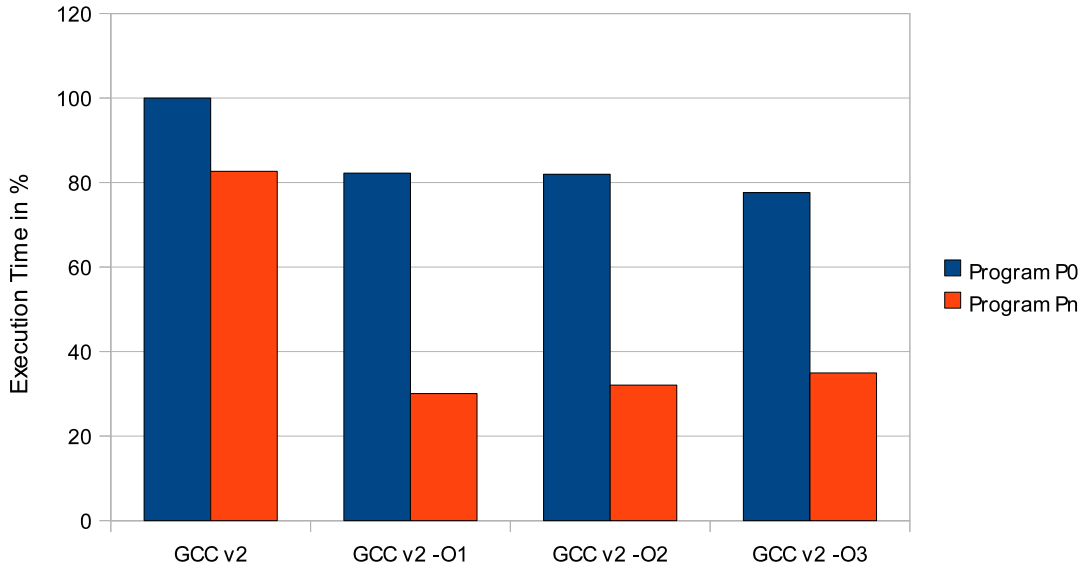
Figure 9.7: Case Study 3: Comparison of the Translated Program States $P_0 ... P_9$ compiled with the GCC v2.



confirm the result of the previous comparison. However, the application of the eighth FermaT transformation has a massive effect on the execution speed of the program especially in combination with compiler optimisations. This transformation decreases the execution time from 466830μs, 386100μs (-O1), 358100μs (-O2) and 356630μs (-O3) of program state P_7 to 415200μs, 151970μs (-O1), 163300μs (-O2) and 173930μs (-O3) of the program state P_8. Further investigation has shown that this transformation eliminates a procedure which swaps the value of two variables. This procedure in turn is called within a loop. The application of the ninth FermaT transformation has once more almost no effect on the execution speed of the program. This means that the program state P_8 and the program state P_9 are almost equally fast. Again, this does not confirm the result of the previous comparison. Figure 9.8 shows the execution time of the translated initial and final program.

The execution time of the final program P_n is only 415270μs, 151030μs (-O1), 161130μs (-O2) and 175570μs (-O3) whereas the execution time of the initial program P_0 is 502400μs, 413100μs (-O1), 411770μs (-O2) and 389970μs (-O3). In other words, the execution speed of the program has been increased by 20.98%, 173.52% (-O1), 155.55% (-O2)

Figure 9.8: Case Study 3: Comparison of the Initial Program P_0 and the Final Program P_n compiled with the GCC v2.

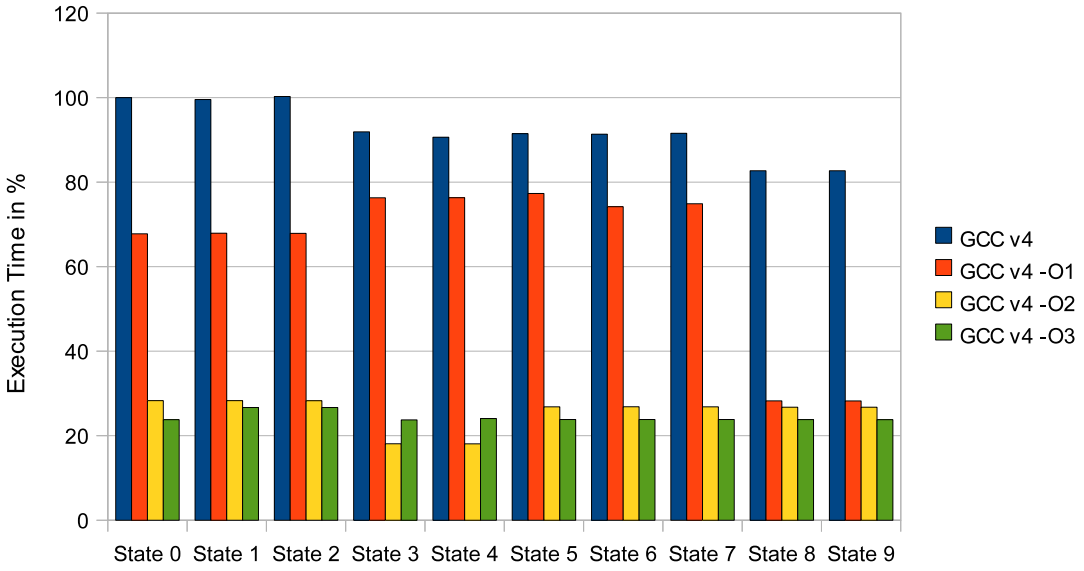


and 122.12% (-O3) in this particular case. On the one hand, this comparison shows that even programs which will be optimised by a particular compiler can benefit from a program transformation process. On the other hand, it also shows how efficient and therefore crucial compiler optimisations are at least in terms of execution speed. For example, the translated final program P_n has an execution time of $175570\mu s$ if it has been compiled with the optimisation option -O3 whereas the same state has an execution time of $415270\mu s$ if it has been compiled without any optimisation at all.

Gnu C Compiler v4

The translated program states $P_0 ... P_9$ of this execution time comparison have been compiled with the GCC v4. This compiler is a successor of the GCC v2. Therefore, it provides three general optimisation options in terms of execution speed as well. Accordingly, there exist four versions of each translated state where the first of these versions has been compiled without any optimisation at all. The other three have been compiled with the optimisation option -O1, -O2 and -O3. Figure 9.9 shows the execution time of the translated states.

Figure 9.9: Case Study 3: Comparison of the Translated Program States $P_0 ... P_9$ compiled with the GCC v4.

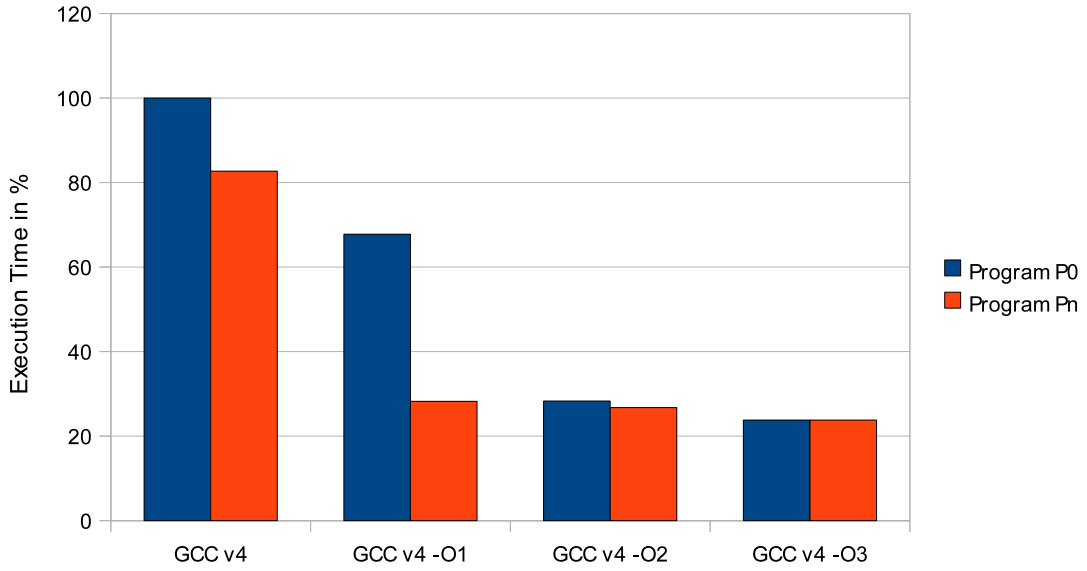


The application of the first FermaT transformation has almost no effect on the execution speed of the program. In other words, the program state P_0 and the program state P_1 are almost equally fast. The same applies to the application of the second FermaT transformation which means that the program state P_1 and the program state P_2 are almost equally fast as well. This confirms the result of the previous comparison where the translated program states have been compiled with the GCC v2. Moreover, it partially confirms the result of the first comparison where the original program states have been interpreted by FermaT 2. The application of the third FermaT transformation has a positive and a negative effect on the execution time depending on the respective optimisation option. It decreases the execution time from 488960μs, 137920μs (-O2) and 130100μs (-O3) of the program state P_2 to 448180μs, 88280μs (-O2) and 115830μs (-O3) of the program state P_3 but it increases the execution time from 331090μs (-O1) of the program state P_2 to 372140μs (-O1) of the program state P_3. This partially confirms the result of the previous comparison but it also shows that there are significant differences between the used compilers. The application of the fourth FermaT transformation in turn has almost no effect on the execution speed of the program which means that the program state P_3 and the program state P_4 are almost equally fast. Once more, this confirms the result of the previous

comparison. However, the application of the fifth FermaT transformation has again a positive and a negative effect on the execution time depending on the respective optimisation option. It decreases the execution time from 117340μs (-O3) of the program state P_4 to 116250μs (-O3) of the program state P_5 but it increases the execution time from 441980μs, 372190μs (-O1) and 88180μs (-O2) of the program state P_4 to 446090μs, 377080μs (-O1) and 130830μs (-O2) of the program state P_5. This confirms neither the result of the previous comparison nor the result of the first comparison. The application of the sixth FermaT transformation has a positive, a negative and no effect on the execution time depending on the respective optimisation option. It decreases the execution time from 446090μs and 377080μs (-O1) of the program state P_5 to 445520μs and 361820μs (-O1) of the program state P_6 but it increases the execution time from 130830μs (-O2) of the program state P_5 to 130940μs (-O2) of the program state P_6. The execution time of the program state P_5 and the program state P_6 which have been compiled with the optimisation option -O3 is exactly the same. This confirms none of the results of the previous comparisons either. The application of the seventh FermaT transformation has again almost no effect on the execution speed of the program whereas the application of the eighth FermaT transformation decreases the execution time from 446510μs, 365210μs (-O1), 130830μs (-O2) and 116250μs (-O3) of the program state P_7 to 403330μs, 137760μs (-O1), 130470μs (-O2) and 116150μs (-O3) of the program state P_8. At least this partially confirms the result of the previous comparison. The application of the ninth FermaT transformation has yet again almost no effect on the execution speed of the program. This means that the program state P_8 and the program state P_9 are almost equally fast which confirms the result of the previous comparison but not the result of the first comparison. Figure 9.10 shows the execution time of the translated initial and final program.

The execution time of the final program P_n is only 403330μs, 137700μs (-O1), 130470μs (-O2) and 116090μs (-O3) whereas the execution time of the initial program P_0 is 487710μs, 330570μs (-O1), 138020μs (-O2) and 116040μs (-O3). In other words, the execution speed of the program has been increased by 20.92%, 40.07% (-O1), 5.79% (-O2) and -0.04% (-O3) in this particular case. On the one hand, this comparison shows again how efficient and therefore crucial compiler optimisations are at least in terms of execution speed. For example, the translated final program P_n has an execution time of 116090μs if it has been compiled with the optimisation option -O3 whereas the same state has an execution time of 403330μs if it has been compiled without any optimisation at all. On the

Figure 9.10: Case Study 3: Comparison of the Initial Program P_0 and the Final Program P_n compiled with the GCC v4.



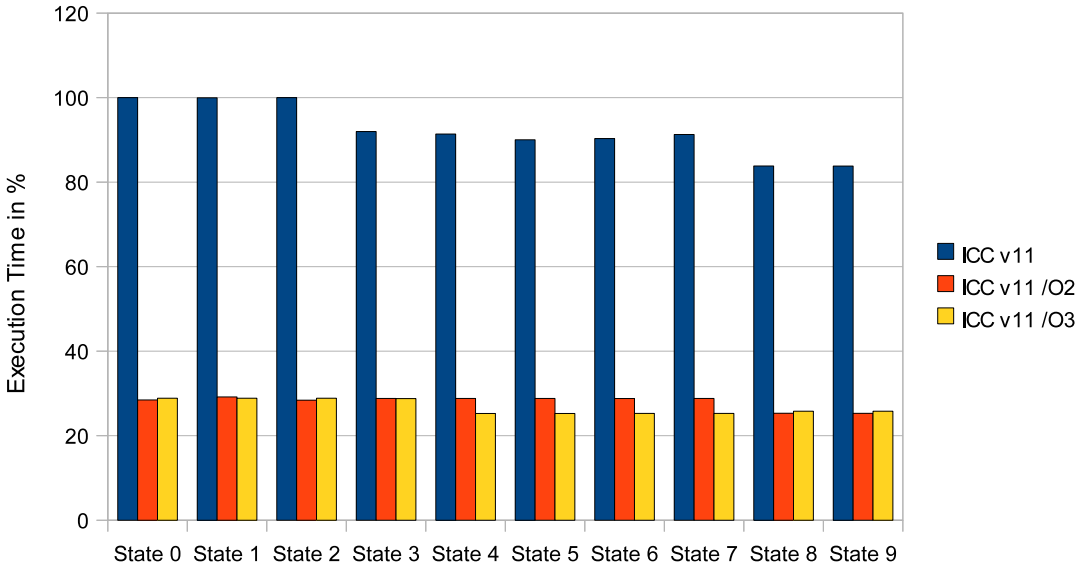
other hand, this comparison reveals that some of the applied FermaT transformations have a negative effect on the execution time depending on the respective optimisation option. For instance, the program state P_4 which has been compiled with the optimisation option -O2 has an execution time of only $88180\mu s$ which is extraordinary fast.

Intel C Compiler v11

The translated program states $P_0 \ldots P_9$ of this execution time comparison have been compiled with the ICC v11. This compiler provides two general optimisation options in terms of execution speed. Accordingly, there exist three versions of each translated state where the first of these versions has been compiled without any optimisation at all. The other two have been compiled with the optimisation option /O2 and /O3. Figure 9.11 shows the execution time of the translated states.

The application of the first FermaT transformation has almost no effect on the execution speed of the program. In other words, the program state P_0 and the program state P_1 are almost equally fast. The same applies to the application of the second FermaT transformation which means that the program state P_1 and the program state P_2 are almost

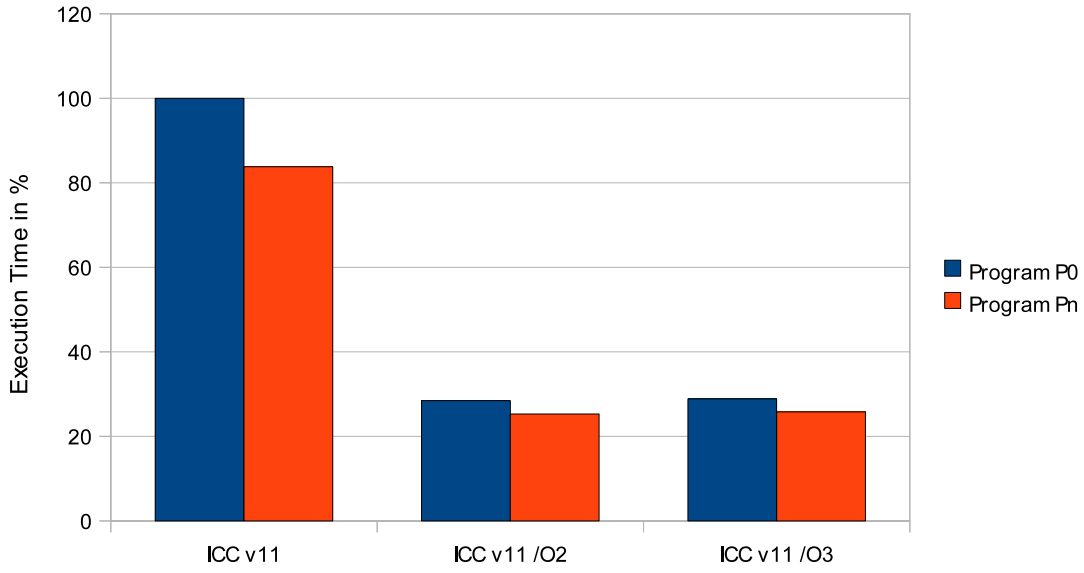
Figure 9.11: Case Study 3: Comparison of the Translated Program States P_0...P_9 compiled with the ICC v11.



equally fast as well. This confirms the result of the previous comparisons where the translated program states have been compiled with the GCC v2 and the GCC v4. Moreover, it partially confirms the result of the first comparison where the original program states have been interpreted by FermaT 2. The application of the third FermaT transformation has a positive and a negative effect on the execution time depending on the respective optimisation option. It decreases the execution time from 483130μs of the program state P_2 to 444380μs of the program state P_3 but it increases the execution time from 137240μs (/O2) and 139480μs (/O3) of the program state P_2 to 139220μs (/O2) and 139060μs (/O3) of the program state P_3. This partially confirms the result of the previous comparisons and shows again that there are significant differences between the used compilers. The application of the fourth FermaT transformation has a positive and no effect on the execution time depending on the respective optimisation option. It decreases the execution time from 444380μs and 139060μs (/O3) of the program state P_3 to 441560μs and 122080μs (/O3) of the program state P_4 whereas the execution time of the program state P_3 and the program state P_4 which have been compiled with the optimisation option /O2 is exactly the same. The application of the following three FermaT transformations in turn has almost no effect on the execution speed of the program. This means that the program state

P_4, the program state P_5 and the program state P_6 are almost equally fast which partially confirms the result of the previous comparisons as well. However, the application of the eighth FermaT transformation has a positive and a negative effect on the execution time depending on the respective optimisation option. It decreases the execution time from 440990μs and 139220μs (/O2) of the program state P_7 to 405000μs and 122290μs (/O2) of the program state P_8 but it increases the execution time from 122140μs (/O3) of the program state P_7 to 124640μs (/O3) of the program state P_8. Again, this partially confirms the result of the previous comparison. The application of the ninth FermaT transformation has yet again almost no effect on the execution speed of the program. This means that the program state P_8 and the program state P_9 are almost equally fast which confirms the result of the previous two comparisons but not the result of the first comparison. Figure 9.12 shows the execution time of the translated initial and final program.

Figure 9.12: Case Study 3: Comparison of the Initial Program P_0 and the Final Program P_n compiled with the ICC v11.



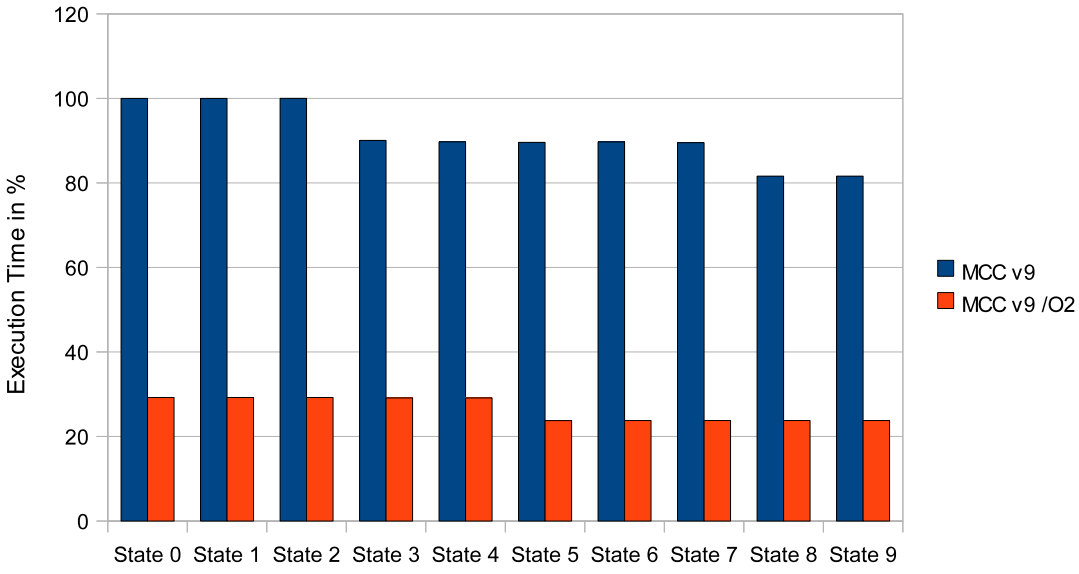
The execution time of the final program P_n is only 404950μs, 122240μs (/O2) and 124640μs (/O3) whereas the execution time of the initial program P_0 is 483180μs, 137500μs (/O2) and 139480μs (/O3). In other words, the execution speed of the program has been increased by 19.32%, 12.48% (/O2) and 11.91% (/O3) in this particular case. Once again,

this comparison shows how efficient and therefore crucial compiler optimisations are at least in terms of execution speed. For example, the translated final program P_n has an execution time of $124640\mu s$ if it has been compiled with the optimisation option /O3 whereas the same state has an execution time of $404950\mu s$ if it has been compiled without any optimisation at all.

Microsoft C Compiler v9

The translated program states $P_0 \ldots P_9$ of this execution time comparison have been compiled with the MCC v9. This compiler provides only one general optimisation option in terms of execution speed. Accordingly, there exist two versions of each translated state where the first of these versions has been compiled without any optimisation at all. The other one has been compiled with the optimisation option /O2. Figure 9.13 shows the execution time of the translated states.

Figure 9.13: Case Study 3: Comparison of the Translated Program States $P_0 \ldots P_9$ compiled with the MCC v9.

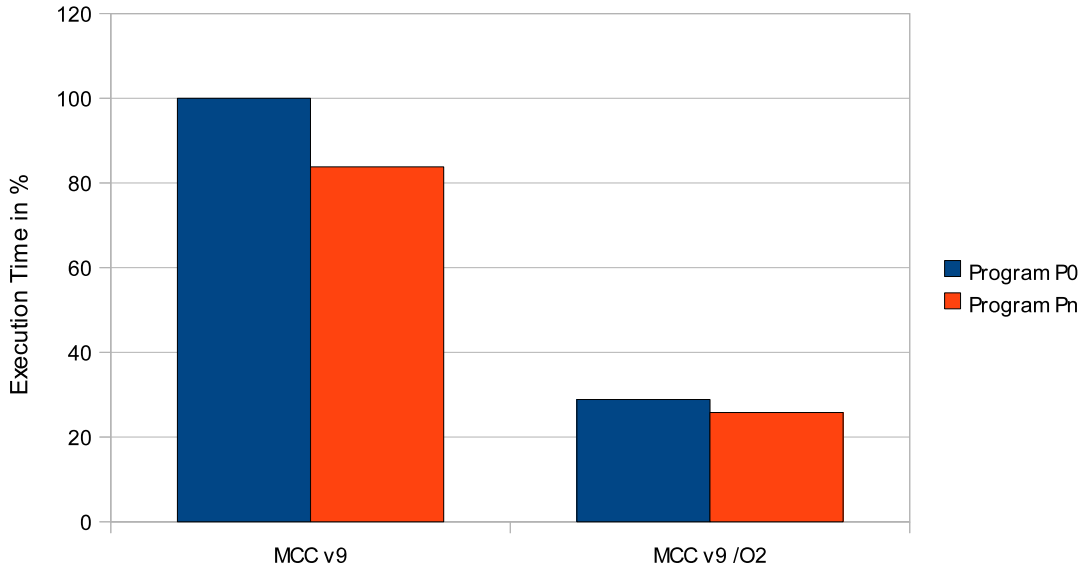


The application of the first FermaT transformation has almost no effect on the execution speed of the program. In other words, the program state P_0 and the program state P_1 are almost equally fast. The same applies to the application of the second FermaT

transformation which means that the program state P_1 and the program state P_2 are almost equally fast as well. This confirms the result of the previous comparisons where the translated program states have been compiled with the GCC v2, the GCC v4 and the ICC v11. Moreover, it partially confirms the result of the first comparison where the original program states have been interpreted by FermaT 2. The application of the third FermaT transformation decreases the execution time from 484480μs and 141610μs (/O2) of the program state P_2 to 436250μs and 141090μs (/O2) of the program state P_3. This partially confirms the result of the previous comparisons and shows once more that there are significant differences between the used compilers. The application of the fourth FermaT transformation in turn has almost no effect on the execution speed of the program which means that the program state P_3 and the program state P_4 are almost equally fast. Again, this confirms the result of the previous comparisons. The application of the fifth FermaT transformation decreases the execution time from 434580μs and 141090μs (/O2) of the program state P_4 to 433960μs and 115110μs (/O2) of the program state P_5. This partially confirms the result of the previous comparisons as well. The application of the following two FermaT transformations has again almost no effect on the execution speed of the program whereas the application of the eighth FermaT transformation has a positive and no effect on the execution time depending on the respective optimisation option. It decreases the execution time from 433590μs of the program state P_7 to 395310μs of the program state P_8 whereas the execution time of the program state P_7 and the program state P_8 which have been compiled with the optimisation option /O2 is exactly the same. Once again, this partially confirms the result of the previous comparison. The application of the ninth FermaT transformation has no effect on the execution speed of the program at all. This means that the program state P_8 and the program state P_9 are almost equally fast which confirms the result of the previous three comparisons but not the result of the first comparison. Figure 9.14 shows the execution time of the translated initial and final program.

The execution time of the final program P_n is only 395310μs and 115160μs (/O2) whereas the execution time of the initial program P_0 is 484380μs and 141610μs (/O2). In other words, the execution speed of the program has been increased by 22.53% and 22.97% (/O2) in this particular case. This comparison shows how efficient and therefore crucial compiler optimisations are even for the MCC v9 which only provides one general optimisation option in terms of execution speed. For example, the translated final program

Figure 9.14: Case Study 3: Comparison of the Initial Program P_0 and the Final Program P_n compiled with the MCC v9.



P_n has an execution time of 115160μs if it has been compiled with the optimisation option /O2 whereas the same state has an execution time of 395310μs if it has been compiled without any optimisation at all.

9.4 Summary

The presented chapter has discussed three medium-scale case studies which describe the use of constraint based transformation theory. The first one has used a very abstract transformation scheme which includes two structure constraints to raise the abstraction level of a program. The definition of the included constraints as well as the development of the abstract transformation scheme has been explained. Furthermore, the application of the transformation scheme and the effect of this application on the given program has been described. The primary intent of this case study was to prove the advantages of the proposed approach compared to search-based approaches even if the maintainer has not much knowledge about transformation theory. On the one hand, this has been achieved due to the prediction technique which was able to reduce the number of transformation sequences within the search space by 48.85%. On the other hand, the evaluation of the

transformation sequences on the basis of transformation effects has totally failed. The second one has used a very concrete transformation scheme which includes two structure constraints to decrease the complexity of a program. The definition of the included constraints as well as the development of the concrete transformation scheme has been explained. Furthermore, the application of the transformation scheme and the effect of this application on the given program has been described. The general aim of this case study was to prove the applicability of the proposed approach on larger programs. This has been achieved because the application of the transformation scheme has been finished in reasonable time on a common PC. The third one has used an abstract transformation scheme which includes four behaviour constraints to decrease the execution time of a program. The definition of the included constraints, the development and the application of the abstract transformation scheme and the effect of this application on the given program has been explained. Furthermore, the execution time of the individual program states has been measured and compared. The main purpose of this case study was to demonstrate the assumption-based approach where constraints are used to achieve an overall target as described in Chapter 4. This has been achieved where the assumptions have been correct in most cases.

Chapter 10

Conclusion and Future Work

Objectives

- To summarise and evaluate the research which has been described in this thesis.
- To discuss the limitations of the presented approach.
- To propose the future work.

This chapter provides a summary of the thesis, evaluates the research questions, discusses the limitations of the proposed approach and presents some suggestions for the future work.

10.1 Summary of the Thesis

The aim of the work which is presented in this thesis is to simplify program transformation processes and to improve the resulting programs. To achieve these ambitious targets, a constraint based program transformation approach which extends the existing FermaT Transformation Engine (FTE) has been presented. This approach is semi-automated and provides the possibility to outline an entire program transformation process on the basis of constraints and transformation schemes. In this context, a constraint is a condition which has to be satisfied at some point during the application of a transformation sequence

whereas a transformation scheme defines the search space which consists of a set of transformation sequences. After the constraints and the scheme have been defined, the system uses a unique knowledge-based prediction technique followed by a particular search tactic to reduce the number of transformation sequences within the search space and to find a transformation sequence which is applicable and which satisfies the given constraints. Moreover, it is possible to describe those transformation schemes with the aid of a formal language which is called Transformation Scheme Description Language (TSDL).

The presented thesis has provided a definition and a classification of constraints for program transformations in Chapter 4. It has discussed capabilities and effects of transformations and their value to define transformation sets in Chapter 5 and in Chapter 6. The modelling of program transformation processes with the aid of transformation schemes which in turn are based on finite automata and the inclusion of constraints into these schemes has been presented in Chapter 6. A formal language to describe transformation schemes and the automated construction of these schemes from the language has been shown in Chapter 6 as well. Furthermore, the thesis has discussed a unique prediction technique which uses the capabilities of transformations, an evaluation of the transformation sequences on the basis of transformation effects and a particular search tactic which is related to linear and tree search tactics in Chapter 7.

The implementation of the constraint based program transformation system has been discussed in Chapter 8 whereas the practical value of this system has been proven with the aid of three medium-scale case studies in Chapter 9. The first one has shown how to raise the abstraction level whereas the second one has shown how to decrease the complexity of a particular program. The third one has shown how to increase the execution speed of a selected program.

10.2 Evaluation

In Chapter 1, a set of research questions were defined. These questions are used in this section to evaluate the proposed approach of this thesis.

1. *Is it possible to model program transformation processes with the aid of maintainer knowledge?*

The modelling of program transformation processes has been proven to be possible in Chapter 6 and in Chapter 9. In terms of this thesis, it has been accomplished by a combination of transformation schemes and constraints. These combination provides a powerful possibility to model program transformation processes and to define program transformation targets. Moreover, the so-called Transformation Scheme Description Language (TSDL) offers a powerful interface to describe transformation schemes and therefore to include the personal knowledge of a maintainer into the program transformation process.

2. *How can the search space of existing search based approaches be restricted?*

   In many cases, the inclusion of maintainer knowledge into the program transformation process significantly restricts the search space. For example, this applies to the second case study of Chapter 9 where the search space contains only 60 transformation sequences although there are ten different FermaT transformations envovled. The collected information to predict the applicability of transformation sequences can also restrict the search space. For example, this applies to the first case study of Chapter 9 where the prediction technique was able to reduce the number of transformation sequences within the search space by 48.85%.

3. *Is the proposed approach able to provide constraint-satisfying program transformation results in reasonable time?*

   On the one hand, the proposed approach is able to provide constraint-satisfying program transformation results in reasonable time as shown in the case studies of Chapter 9. On the other hand, this depends a lot on the size of the given program, on the developed transformation scheme and on the defined constraints.

4. *How can constraints be integrated into program transformation theory?*

   The presented approach proposes the modelling of program transformation processes with the aid of so-called transformation schemes which have been discussed in Chapter 6. These schemes are explicitly developed to support the integration of constraints. Therefore, it is not only possible to define constraints which have to be satisfied at the end of the program transformation process. It is also possible to determine at which state within the process these constraints have to be satisfied. For example, this allows the application of a milestone strategy where a constraint has to be satisfied to guide the satisfaction of another constraint.

5. *What are the advantages of a constraint based program transformation approach against a common program transformation approach?*

   The advantages of this approach are as follows:

   - Constraint based program transformation theory provides a powerful solution to describe, capture and analyse entire program transformation processes. It allows to engineer such a process rather than applying one transformation after another.
   - The program transformation process has been simplified because it is not necessary that a maintainer knows the effect of each particular transformation as it is the case with the current FermaT Transformation Engine. Also, it is not necessary to comprehend and measure the WSL code after the application of each transformation.
   - The application of the FermaT transformations has been automated. This is because the entire program transformation process as well as all program transformation targets can be defined before even one FermaT transformation has been applied. Once the definition has been finished, the application of the envolved FermaT transformations does not require any interaction of the maintainer.
   - The program transformation process can be modelled with the aid of a formal language. Therefore, the maintainer has a powerful interface to include his personal knowledge into the process. This is optional and leads to a restricted search space.
   - The effect of a FermaT transformation in relation to a particular constraint can be predicted to a certain degree which leads to a restricted search space as well.
   - The developed transformation schemes and the defined constraints can be reutilised for other program transformation processes.

10.3 Limitations

The proposed approach of this thesis has a lot of potential to improve the program transformation processes. Furthermore, it has proven its practical value with the aid of three

medium-scale case studies. Nevertheless, there are limitations which can be summarised as follows:

1. **Classification of Constraints**

   Chapter 4 of this thesis provides an extensive classification of constraints which primarily serves as an overview. However, this classification is neither complete nor have all constraints which appear in this classification been used in combination with the constraint based program transformation system.

2. **Search Space**

   One of the main limitations is the search-space which can still be huge if the transformation scheme is very abstract. The modelling of program transformation processes and the use of a prediction technique are often able to restrict the search-space which has been shown in Chapter 9 but in some cases a search is simply infeasible.

3. **Maintainer Knowledge**

   The proposed approach of this thesis uses maintainer knowledge to restrict the search space. Therefore, an entire program transformation process will possibly fail if this knowledge is incorrect.

4. **Behavioural Constraints**

   The approach of behavioural constraints is partially based on assumptions which has been discussed in Chapter 4. It can not be proven that this approach will lead to the desired target in every case.

5. **Transformation Sequence Evaluation Failure**

   The evaluation of the transformation sequences on the basis of transformation effects which has been discussed in Chapter 7 has failed in the three case studies of Chapter 9.

10.4 Future Work

The proposed approach of this thesis describes a constraint based program transformation system to simplify and improve the task of software adaption and optimisation. It provides

a solid foundation for further research but it is not the terminus. The following future work can be pursued based on the presented work:

1. **Extension of the Constraint Classification**

   Chapter 4 of this thesis has discussed an extensive classification of constraints. This classification contains various structural and behavioural constraints which are important for a lot of program transformation processes. However, a lot of constraints which appear in this classification are only mentioned or described to provide an overview and are not implemented yet. Furthermore, the classification is not complete and there certainly exist a lot more constraints which can be quite useful.

2. **Capabilities and Effects of Transformation Sequences**

   Chapter 5 discusses transformation capabilities and effects. These can be extracted from the individual FermaT transformations. Afterwards, it is possible to use the captured information not only for comprehension purposes but also for the prediction technique which has been discussed in Chapter 7 or the evaluation of the transformation sequences which has been discussed in Chapter 7 as well. To take interplay effects into account, it might be an idea to extract capabilities and effects not only from individual transformations but also from entire transformation sequences.

3. **Prediction Technique Improvements**

   The prediction technique which has been discussed in Chapter 7 is very important for constraint based program transformation processes. Moreover, further enhancements are necessary because the technique has its problems with well defined transformation schemes. This has been discussed in Chapter 9. For example, it is conceivable to improve the mathematical model of this technique to the effect that it will take not only AST types into account which have been added by the FermaT transformations but also AST types which have been removed.

4. **Transformation Sequence Evaluation Improvements**

   The evaluation of the transformation sequences on the basis of transformation effects which has been discussed in Chapter 7 is very important for constraint based program transformation processes as well. The failure of this technique causes often a slower application of the respective transformation scheme which has been

discussed in Chapter 9. Therefore, further improvements are necessary which can be achieved by an update of the formula R_S. This formula is used to create a transformation sequence rating.

5. **Search Tactic Improvements**

   The search tactic which has been discussed in Chapter 7 is even more important for constraint based program transformation processes. Moreover, there is a lot of room for enhancements. For example, it is conceivable to use hill-climbing or genetic algorithms for further improvements.

6. **Adaptation of the FermaT Transformations**

   The FermaT transformations are very extensive and provide a lot of program transformation possibilities. Nevertheless, they are not optimised for an automated application. The effect of some transformations is redundant and there are a lot of exceptions which have to be considered. For example, some transformations do not have a proper applicability condition. This makes a systematic application of FermaT transformations difficult. For this reason, it is required to adapt and enhance the individual FermaT transformations for the constraint based program transformation system.

7. **Constraint Based Program Transformation Processes on Distributed Systems**

   A constraint based program transformation process often envolves the application of thousands of independent FermaT transformations. This time consuming task seems to be extremely suitable to be executed on distributed systems. Therefore, a research project on this promising field has already been started [7].

References

[1] A.V. Aho, R. Sethi, and J.D. Ullmann. *Compilerbau*. Oldenbourg, second edition, December 1999.

[2] J.A. Anderson. *Automata Theory with Modern Applications*. Cambridge University Press, first edition, July 2006.

[3] A.W. Apple. *Modern Compiler Implementation in ML*. Cambridge University Press, first edition, March 1998.

[4] R.S. Arnold. Software restructuring. In *Proceedings of the IEEE*. IEEE Computer Society, April 1989.

[5] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*. Mathematical Centre Amsterdam, first edition, April 1980.

[6] R.J.R. Back and J.v. Right. Refinement concepts formalised in higher order logic. *Formal Aspects of Computing*, 2:247–272, September 1989.

[7] P. Bartels. Clustering techniques on transformation systems. Technical report, Software Technology Research Laboratory, De Montfort University, November 2007.

[8] F.L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner, H. Partsch, P. Pepper, and H. Wössner. Towards a wide spectrum language to support program specification and program development. *ACM SIGPLAN Notices*, 13:15–24, December 1978.

[9] I.D. Baxter. *Transformational Maintenance by Reuse of Design Histories*. PhD thesis, Department of Information and Computer Science, University of California, November 1990.

[10] I.D. Baxter. Branch coverage for arbitrary languages made easy. Technical report, Semantic Designs, january 2002.

[11] I.D. Baxter, C. Pidgeon, and M. Mehlich. Dms: Program transformations for practical scalable software evolution. In *IEEE International Conference on Software Maintenance*. IEEE Computer Society, May 2004.

[12] M. Bravenboer, A.v. Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. Technical report, Institute of Information and Computing Sciences, Utrecht University, June 2005.

[13] F.P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, first edition, January 1975.

[14] R. Bruns and C. Möbus. *Wissensbasierte Genetische Algorithmen: Integration von Genetischen Algorithmen und Constraint Programmierung zur Lösung kombinatorischer Optimierungsprobleme*. Akademische Verlagsgesellschaft Aka, first edition, July 1996.

[15] J.N. Buxton, P. Naur, and B.R. Randall. Software engineering, concepts and techniques. In *NATO Software Engineering Conferences*. NATO Science Committee, January 1976.

[16] E.J. Byrne. A conceptual foundation for software re-engineering. In *IEEE International Conference on Software Maintenance*. IEEE Computer Society, November 1992.

[17] C.H. Chang. From regular expressions to dfa's using compressed nfa's. Technical report, Courant Institute of Mathematical Sciences, New York University, October 1992.

[18] F. Chen, S. Li, and W.C.C. Chu. Feature analysis for service-oriented reengineering. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*. IEEE Computer Society, December 2005.

[19] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Transactions on Software Engineering*, 7:13–17, January 1990.

[20] K.C. Chisolm and J.C. Lisonbee. The use of computer language compilers in legacy code migration. In *IEEE Systems Readiness Technology Conference*. IEEE Computer Society, August 1999.

[21] K.D. Cooper, P.J. Schielke, and D. Subramanian. Optimising for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*. ACM SIGPLAN, May 1999.

[22] K.D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23:7–22, August 2002.

[23] T. DeMarco. *Structured Analysis and System Specification*. Prentice-Hall, first edition, June 1979.

[24] T. DeMarco. *Controlling Software Projects: Management, Measurement and Estimation*. Prentice-Hall, first edition, November 1982.

[25] E.W.K.C. Denney. *A Theory of Program Refinement*. PhD thesis, University of Edinburgh, June 1998.

[26] E.W. Dijkstra. The humble programmer. *ACM*, 15:859–866, October 1972.

[27] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, August 1975.

[28] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, first edition, January 1976.

[29] Jacques Eloff. Software restructuring: Implementing a code abstraction transformation. In *Proceedings of the South African Institute of Computer Scientists*. ACM SAICSIT, September 2002.

[30] D. Fatiregun, M. Harman, and R.M. Hierons. Search based transformations. In *Genetic and Evolutionary Computation Conference*. American Association for Artificial Intelligence, July 2003.

[31] D. Fatiregun, M. Harman, and R.M. Hierons. Evolving transformation sequences using genetic algorithms. In *IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, September 2004.

[32] Y.A. Feldman and D.A. Friedman. Portability by automatic translation: A large-scale case study. In *Knowledge-Based Software Engineering Conference*. IEEE Computer Society, November 1995.

[33] R.W. Floyd. Assigning meanings to programs. *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, 19:19–31, April 1967.

[34] F. Geiselbrechtinger, W. Hesse, B. Krieg-Brückner, and H. Scheidig. The basic layer of a wide spectrum language. *Lecture Notes in Computer Science*, 3:188–197, October 1973.

[35] F. Geiselbrechtinger, W. Hesse, B. Krieg-Brückner, and H. Scheidig. Recent developments concerning wide spectrum languages. In *Working Conference on Machine-Oriented Higher-Level Languages*. International Federation for Information Processing, August 1974.

[36] K.J. Gough. *Multi Language - Multi Target Compiler Development: Evolution of the Gardens Point Compiler Project*. Springer, first edition, March 1997.

[37] P.A.V. Hall. *Software Reuse and Reverse Engineering in Practise*. Chapman-Hall, first edition, March 1992.

[38] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, fourth edition, September 2006.

[39] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, October 1969.

[40] J.E. Hopcroft, R. Motwani, and J.D. Ullmann. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, November 2000.

[41] H. Hua. Software evolution based on software architecture. In *International Conference on Computer and Information Technology*. IEEE Computer Society, September 2004.

[42] Intel. *Quick-Reference Guide to Optimization with Intel Compilers Version 11*, November 2008.

[43] ISO / IEC. *C - International Standard Specification*, September 2007.

[44] C. Jones. Backfiring: Converting lines of code to function points. *Computer - Innovative Technology for Computing Professionals*, 28:87–88, November 1995.

[45] C.R. Karp. *Languages with Expressions of Infinite Length*. North-Holland Publishing Company, first edition, December 1964.

[46] R. Kelsey, W. Clinger, and J. Rees. Fifth revised report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33:26–76, February 1998.

[47] C.F. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25:493–509, July 1999.

[48] B.W. Kernighan and R. Pike. Regular expressions: Languages, algorithms and software. Technical report, Bell Laboratories, April 1999.

[49] M. Ladkau. Fermat unified modeling language. Technical report, Department of Electrical Engineering and Computer Science, University of Applied Sciences Emden, February 2006.

[50] M. Ladkau. A type transformation system for legacy applications. Technical report, Software Technology Research Laboratory, De Montfort University, April 2007.

[51] M. Ladkau. *A Wide Spectrum Type System*. PhD thesis, Software Technology Research Laboratory, De Montfort University, April 2009.

[52] M. Ladkau, F. Chen, S. Li, and S. Natelberg. The fermat transformation engine. Technical report, Software Technology Research Laboratory, De Montfort University, October 2006.

[53] Leslie Lamport. Win and sin: Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems*, 12:396–428, July 1990.

[54] M.M. Lehman. The programming process. Technical report, IBM Research Centre Yorktown Heights, September 1969.

[55] M.M. Lehman. Programs, cities, students - limits to growth? *Imperial College of Science and Technology, Inaugural Lecture Series*, 9:211–229, May 1974.

[56] M.M. Lehman. Laws of program evolution - rules and tools for programming management. In *Proceedings of the Infotech State of the Art Conference - Why Software Projects Fail*. Pergamon Infotech, April 1978.

[57] M.M. Lehman. On understanding laws, evolution and conservation in the large program life-cycle. *Journal of Systems and Software*, 1:213–228, January 1980.

[58] M.M. Lehman. Uncertainty in computer application and its control through the engineering of software. *Journal of Software Maintenance: Research and Practise*, 1:3–27, September 1989.

[59] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Tursky. Metrics and laws of software evolution - the nineties view. In *International Software Metrics Symposium*. IEEE Computer Society, November 1997.

[60] S. Li. *A Program Transformation Step Prediction Based Re-engineering Approach*. PhD thesis, Software Technology Research Laboratory, De Montfort University, October 2007.

[61] R. Lämmel and C. Verhoef. Cracking the 500 language problem. Technical report, Department of Mathematics and Computer Science, Free University of Amsterdam, September 2001.

[62] E.G. Manes. *Predicate Transformer Semantics*. Cambridge University Press, first edition, October 2004.

[63] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, December 1976.

[64] R. Millham. An investigation: Re-engineering sequential procedure-driven software into object-oriented event-driven software through uml diagrams. In *International Computer Software and Applications Conference*. IEEE Computer Society, August 2002.

[65] S. Natelberg. Fermat documentation for legacy systems. Technical report, Department of Electrical Engineering and Computer Science, University of Applied Sciences Emden, April 2006.

[66] S. Natelberg. Constraint based program transformation theory. Technical report, Software Technology Research Laboratory, De Montfort University, November 2007.

[67] D.A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtyping. Technical report, Department of Computer Science, Stevens Institute of Technology, November 1998.

[68] P. Norvig and S. Russell. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, second edition, January 2003.

[69] N. Popov. Verification by using a weakest precondition strategy. Technical report, Research Institute for Symbolic Computation, Hagenberg, February 2003.

[70] J. Reinfelds. Programming as an engineering discipline. In *Frontiers in Education Conference*. IEEE Computer Society, November 2002.

[71] J. Scott. The e-business hat trick - adaptive enterprises, adaptable software, agile it professionals. *Cutter IT Journal*, 13:7–12, April 2000.

[72] R.C. Seacord, D. Plakosh, and G.A. Lewis. *Modernising Legacy Systems: Software Technologies, Engineering Processes and Business Practices*. Addison-Wesley, first edition, February 2003.

[73] H. Sneed and C. Verhoef. Re-engineering the corporation - a manifesto for it evolution. Technical report, Department of Mathematics and Computer Science, Free University of Amsterdam, May 2001.

[74] Software Technology Research Laboratory. *WSL Programmer's Reference Manual*, July 2008.

[75] R.M. Stallman. *Using the GNU Compiler Collection*. GCC Developer Community, May 2004.

[76] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, second edition, September 1981.

[77] Sun Microsystems. *Java SE Development Kit Documentation 6*, December 2006.

[78] R.D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19:437–453, August 1976.

[79] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, June 1968.

[80] W.M. Ulrich. *Legacy Systems: Transformation Strategies*. Prentice-Hall, first edition, June 2002.

[81] E. Visser. Stratego: A language for program transformation based on rewriting strategies. Technical report, Institute of Information and Computing Sciences, Utrecht University, May 2001.

[82] E. Visser. Program transformation with stratego/xt: Rules, strategies, tools and systems. Technical report, Institute of Information and Computing Sciences, Utrecht University, February 2004.

[83] M.P. Ward. *Proving Program Refinements and Transformations*. PhD thesis, St. Annes College, Oxford University, June 1989.

[84] M.P. Ward. Using formal transformations to construct a component repository. *Software Reuse: The European Approach*, 7:1–11, February 1991.

[85] M.P. Ward. Reverse engineering through formal transformation: Knuth's polynomial addition algorithm. *The Computer Journal*, 37:795–813, September 1994.

[86] M.P. Ward. A definition of abstraction. *Journal of Software Maintenance: Research and Practise*, 7:443–450, November 1995.

[87] M.P. Ward. Assembler to c migration using the fermat transformation system. In *IEEE International Conference on Software Maintenance*. IEEE Computer Society, August 1999.

[88] M.P. Ward. The fermat assembler re-engineering workbench. In *IEEE International Conference on Software Maintenance*. IEEE Computer Society, November 2001.

[89] M.P. Ward. Program slicing via fermat transformations. In *International Computer Software and Applications Conference*. IEEE Computer Society, August 2002.

[90] M.P. Ward. Pigs from sausages? re-engineering from assembler to c via fermat transformations. *Science of Computer Programming - Special Issue on Program Transformation*, 52:213–251, August 2004.

[91] M.P. Ward and K.H. Bennett. A practical program transformation system. In *Working Conference on Reverse Engineering*. IEEE Computer Society, May 1993.

[92] M.P. Ward and K.H. Bennett. Formal methods to aid the evolution of software. *International Journal of Software Engineering and Knowledge Engineering*, 5:25–47, January 1995.

[93] M.P. Ward and K.H. Bennett. Recursion removal / introduction by formal transformation: An aid to program development and program comprehension. *The Computer Journal*, 42:650–673, September 1999.

[94] M.P. Ward and H. Zedan. Meta-wsl and meta-transformations in the fermat transformation system. In *International Computer Software and Applications Conference*. IEEE Computer Society, July 2005.

[95] M.P. Ward and H. Zedan. Analysing and abstracting legacy assembler code via conditioned semantic slicing. Technical report, Software Technology Research Laboratory, De Montfort University, September 2006.

[96] M.P. Ward and H. Zedan. Slicing as a program transformation. *ACM Transactions on Programming Languages and Systems*, 29:1–52, April 2007.

[97] M.P. Ward, H. Zedan, and T. Hardcastle. Legacy assembler re-engineering and migration. In *IEEE International Conference on Software Maintenance*. IEEE Computer Society, September 2004.

[98] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.

[99] H. Yang and M.P. Ward. *Successful Evolution of Software Systems*. Artech House Publishers, first edition, January 2003.

[100] X. Zhang, M. Munro, M. Harman, and L. Hu. Mechanised operational semantics of wsl. In *IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, October 2002.

Appendix A

AST Types in WSL

A WSL program is internally handled as Abstract Syntax Tree (AST) which consists of a collection of general types, group types and specific types. The root of such an AST is always the group type T_Statements. Each type contains a precise definition of the subtypes which are possible. Furthermore, each of the specific types is associated with a particular general type. The general types in turn can only contain specific types which are associated to them.

A general type within an AST contains always two other general types, one general type and one group type or one associated specific type. A group type can be considered as package of some particular general types. Each general type except the T_Condition and the T_Name has a related group type. In fact, these behave like a set of general types which has been collapsed down. Specific types in turn can contain general types as well as group types. It is also possible that they contain nothing at all.

The described structure is quite helpful to implement FermaT transformations which are basically operations on the AST. The following tables provide an overview about the relation of the particular general types, group types and specific types in WSL [74].

The table shows the general types and the group types in WSL. A general type in an AST always contains another general type or a specific type whereas a group type can contain an unlimited number of a particular general type.

ID	General Type	Subtypes	WSL syntax	ID	Group Type	Subtypes	WSL syntax
*A*1	T_Action	*A*8, *B*7	ACTION ... == ... END	*B*1	T_Actions	*A*1	
*A*2	T_Assign	*A*7, *A*5	... := ...	*B*2	T_Assigns	*A*2	..., ...
*A*3	T_Condition						
*A*4	T_Definition			*B*3	T_Definitions	*A*4	
*A*5	T_Expression			*B*4	T_Expressions	*A*5	..., ...
*A*6	T_Guarded	*A*3, *B*7		*B*5	T_Guardeds	*A*6	
*A*7	T_Lvalue			*B*6	T_Lvalues	*A*7	..., ...
*A*8	T_Name		...				
*A*9	T_Statement			*B*7	T_Statements	*A*9	...; ...

The table shows the Specific Types in WSL which are associated with a particular general type.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*1	T_Abort	T_Statement		ABORT
*C*2	T_Abs	T_Expression	*A*5	ABS (...)
*C*3	T_Action_Int_Any	T_Action	*A*5	
*C*4	T_Action_Int_One	T_Action	*A*5	
*C*5	T_Action_Pat_Any	T_Action		+ ...
*C*6	T_Action_Pat_Many	T_Action		* ...
*C*7	T_Action_Pat_One	T_Action		? ...
*C*8	T_Action_Val_Any	T_Action		
*C*9	T_Action_Val_One	T_Action		
*C*10	T_And	T_Condition	*A*3, *A*3	... AND ...
*C*11	T_Aref	T_Expression	*A*5, *B*4	... [...]
*C*12	T_Aref_Lvalue	T_Lvalue	*A*7, *B*4	... (...)
*C*13	T_Array	T_Expression	*A*5, *A*5	ARRAY (..., ...)
*C*14	T_Assert	T_Statement	*A*3	{...}
*C*15	T_Assignment	T_Statement	*A*2	
*C*16	T_Assign_Int_Any	T_Assign	*A*5	
*C*17	T_Assign_Int_One	T_Assign	*A*5	

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*18	T_Assign_Pat_Any	T_Assign		+ ...
*C*19	T_Assign_Pat_Many	T_Assign		* ...
*C*20	T_Assign_Pat_One	T_Assign		? ...
*C*21	T_Assign_Val_Any	T_Assign		
*C*22	T_Assign_Val_One	T_Assign		
*C*23	T_Ateach_Cond	T_Statement	*B*7	ATEACH Condition DO ... OD
*C*24	T_Ateach_Expn	T_Statement	*B*7	ATEACH Expression DO ... OD
*C*25	T_Ateach_Global_Var	T_Statement	*B*7	ATEACH Global Variable DO ... OD
*C*26	T_Ateach_Lvalue	T_Statement	*B*7	ATEACH Lvalue DO ... OD
*C*27	T_Ateach_NAS	T_Statement	*B*7	ATEACH NAS DO ... OD
*C*28	T_Ateach_Stat	T_Statement	*B*7	ATEACH Statement DO ... OD
*C*29	T_Ateach_Stats	T_Statement	*B*7	ATEACH Statements DO ... OD
*C*30	T_Ateach_STS	T_Statement	*B*7	ATEACH STS DO ... OD
*C*31	T_Ateach_TS	T_Statement	*B*7	ATEACH Terminal Statement DO ... OD
*C*32	T_Ateach_TSs	T_Statement	*B*7	ATEACH Terminal Statements DO ... OD
*C*33	T_Ateach_Variable	T_Statement	*B*7	ATEACH Variable DO ... OD
*C*34	T_A_Proc_Call	T_Statement	*A*8, *B*4, *B*6	!P ... (... VAR ...)
*C*35	T_A_S	T_Statement	*A*8, *B*1	ACTIONS ...: ... ENDACTIONS

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*36	T_BFunct	T_Definition	*A*8, *B*6, *B*2, *A*3	BFUNCT ... ?(...) == VAR <...>: (...) END
*C*37	T_BFunct_Call	T_Condition	*A*8, *B*4	... ?(...)
*C*38	T_Butlast	T_Expression	*A*5	BUTLAST (...)
*C*39	T_Call	T_Statement		CALL ...
*C*40	T_Comment	T_Statement		C:"..."
*C*41	T_Concat	T_Expression	*A*5, *A*5	... ++ ...
*C*42	T_Cond	T_Statement	*B*5	IF ... THEN ... (ELSIF ...)+ (ELSE ...)? FI
*C*43	T_Cond_Int_Any	T_Condition	*A*5	
*C*44	T_Cond_Int_One	T_Condition	*A*5	
*C*45	T_Cond_Pat_Any	T_Condition		* ...
*C*46	T_Cond_Pat_Many	T_Condition		+ ...
*C*47	T_Cond_Pat_One	T_Condition		? ...
*C*48	T_Cond_Place	T_Condition		$Condition$
*C*49	T_Cond_Val_Any	T_Condition		
*C*50	T_Cond_Val_One	T_Condition		
*C*51	T_Defn_Int_Any	T_Definition	*A*5	
*C*52	T_Defn_Int_One	T_Definition	*A*5	
*C*53	T_Defn_Pat_Any	T_Definition		* ...

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*54	T_Defn_Pat_Many	T_Definition		+ ...
*C*55	T_Defn_Pat_One	T_Definition		? ...
*C*56	T_Defn_Val_Any	T_Definition		
*C*57	T_Defn_Val_One	T_Definition		
*C*58	T_Div	T_Expression	*A*5, *A*5	... DIV ...
*C*59	T_Divide	T_Expression	*A*5, *A*5	... / ...
*C*60	T_D_Do	T_Statement	*B*5	D_DO ... -> ...([] ... -> ...)* OD
*C*61	T_D_IF	T_Statement	*B*5	D_IF ... -> ...([] ... -> ...)* FI
*C*62	T_Empty	T_Condition	*A*5	EMPTY ?(...)
*C*63	T_Equal	T_Condition	*A*5, *A*5	... = ...
*C*64	T_Error	T_Statement	*B*4	ERROR (...)
*C*65	T_Even	T_Condition	*A*5	EVEN ?(...)
*C*66	T_Exists	T_Condition	*B*6, *A*3	EXISTS <...>; ... END
*C*67	T_Exit	T_Statement		EXIT (...)
*C*68	T_Expn_Int_Any	T_Expression	*A*5	
*C*69	T_Expn_Int_One	T_Expression	*A*5	
*C*70	T_Expn_Pat_Any	T_Expression		* ...
*C*71	T_Expn_Pat_Many	T_Expression		+ ...

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*72	T_Expn_Pat_One	T_Expression		? ...
*C*73	T_Expn_Place	T_Expression		$Expn$
*C*74	T_Expn_Val_Any	T_Expression		
*C*75	T_Expn_Val_One	T_Expression		
*C*76	T_Exponent	T_Expression	*A*5, *A*5	... ** ...
*C*77	T_False	T_Condition		FALSE
*C*78	T_Fill2_Action	T_Expression	*A*1	FILL2 Action ... ENDFILL
*C*79	T_Fill2_Assign	T_Expression	*A*2	FILL2 Assign ... ENDFILL
*C*80	T_Fill2_Assigns	T_Expression	*B*2	FILL2 Assigns ... ENDFILL
*C*81	T_Fill2_Cond	T_Expression	*A*3	FILL2 Condition ... ENDFILL
*C*82	T_Fill2_Defn	T_Expression	*A*4	FILL2 Definition ... ENDFILL
*C*83	T_Fill2_Defns	T_Expression	*B*3	FILL2 Definitions ... ENDFILL
*C*84	T_Fill2_Expn	T_Expression	*A*5	FILL2 Expression ... ENDFILL
*C*85	T_Fill2_Expns	T_Expression	*B*4	FILL2 Expressions ... ENDFILL
*C*86	T_Fill2_Guarded	T_Expression	*A*6	FILL2 Guarded ... ENDFILL
*C*87	T_Fill2_Lvalue	T_Expression	*A*7	FILL2 Lvalue ... ENDFILL
*C*88	T_Fill2_Lvalues	T_Expression	*B*6	FILL2 Lvalues ... ENDFILL
*C*89	T_Fill2_Stat	T_Expression	*A*9	FILL2 Statement ... ENDFILL

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*90	T_Fill2_Stats	T_Expression	*B*7	FILL2 Statements ... ENDFILL
*C*91	T_Fill_Action	T_Expression	*A*1	FILL Action ... ENDFILL
*C*92	T_Fill_Assign	T_Expression	*A*2	FILL Assign ... ENDFILL
*C*93	T_Fill_Assigns	T_Expression	*B*2	FILL Assigns ... ENDFILL
*C*94	T_Fill_Cond	T_Expression	*A*3	FILL Condition ... ENDFILL
*C*95	T_Fill_Defn	T_Expression	*A*4	FILL Definition ... ENDFILL
*C*96	T_Fill_Defns	T_Expression	*B*3	FILL Definitions ... ENDFILL
*C*97	T_Fill_Expn	T_Expression	*A*5	FILL Expression ... ENDFILL
*C*98	T_Fill_Expns	T_Expression	*B*4	FILL Expressions ... ENDFILL
*C*99	T_Fill_Guarded	T_Expression	*A*6	FILL Guarded ... ENDFILL
*C*100	T_Fill_Lvalue	T_Expression	*A*7	FILL Lvalue ... ENDFILL
*C*101	T_Fill_Lvalues	T_Expression	*B*6	FILL Lvalues ... ENDFILL
*C*102	T_Fill_Stat	T_Expression	*A*9	FILL Statement ... ENDFILL
*C*103	T_Fill_Stats	T_Expression	*B*7	FILL Statements ... ENDFILL
*C*104	T_Final_Seg	T_Expression	*A*5, *A*5	... [... ..]
*C*105	T_Final_Seg_Lvalue	T_Lvalue	*A*7, *A*5	... (... ..)
*C*106	T_Floop	T_Statement	*B*7	DO ... OD
*C*107	T_For	T_Statement	*A*7, *A*5, *A*5, *A*5, *B*7	FOR ... := ... TO ... STEP ... DO ... OD

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*108	T_Forall	T_Condition	*B*6, *A*3	FORALL <...>; ... END
*C*109	T_Foreach_Cond	T_Statement	*B*7	FOREACH Condition DO ... OD
*C*110	T_Foreach_Expn	T_Statement	*B*7	FOREACH Expression DO ... OD
*C*111	T_Foreach_Global_Var	T_Statement	*B*7	FOREACH Global Variable DO ... OD
*C*112	T_Foreach_Lvalue	T_Statement	*B*7	FOREACH Lvalue DO ... OD
*C*113	T_Foreach_NAS	T_Statement	*B*7	FOREACH NAS DO ... OD
*C*114	T_Foreach_Stat	T_Statement	*B*7	FOREACH Statement DO ... OD
*C*115	T_Foreach_Stats	T_Statement	*B*7	FOREACH Statements DO ... OD
*C*116	T_Foreach_STS	T_Statement	*B*7	FOREACH STS DO ... OD
*C*117	T_Foreach_TS	T_Statement	*B*7	FOREACH Terminal Statement DO ... OD
*C*118	T_Foreach_TSs	T_Statement	*B*7	FOREACH Terminal Statements DO ... OD
*C*119	T_Foreach_Variable	T_Statement	*B*7	FOREACH Variable DO ... OD
*C*120	T_For_In	T_Statement	*A*7, *A*5, *B*7	FOR ... IN ... DO ... OD
*C*121	T_Frac	T_Expression	*A*5	FRAC (...)
*C*122	T_Funct	T_Definition	*A*8, *B*6, *B*2, *A*5	FUNCT ... (...) == VAR <...>: (...) END
*C*123	T_Funct_Call	T_Expression	*A*8, *B*4	... (...)
*C*124	T_Get	T_Expression	*A*5, *A*5	...^...
*C*125	T_Gethash	T_Expression	*A*5, *A*5	... . (...)

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*126	T_Get_n	T_Expression	*A*5, *A*5	...^...
*C*127	T_Greater	T_Condition	*A*5, *A*5	... > ...
*C*128	T_Greater_Eq	T_Condition	*A*5, *A*5	... >= ...
*C*129	T_Guarded_Int_Any	T_Guarded	*A*5	
*C*130	T_Guarded_Int_One	T_Guarded	*A*5	
*C*131	T_Guarded_Pat_Any	T_Guarded		+ ...
*C*132	T_Guarded_Pat_Many	T_Guarded		* ...
*C*133	T_Guarded_Pat_One	T_Guarded		? ...
*C*134	T_Guarded_Val_Any	T_Guarded		
*C*135	T_Guarded_Val_One	T_Guarded		
*C*136	T_Hashtable	T_Expression		HASH_TABLE
*C*137	T_Head	T_Expression	*A*5	HEAD (...)
*C*138	T_Ifmatch2_Action	T_Statement	*A*1, *B*7, *B*7	IFMATCH2 Action ... THEN ... ELSE ... ENDMATCH
*C*139	T_Ifmatch2_Assign	T_Statement	*A*2, *B*7, *B*7	IFMATCH2 Assign ... THEN ... ELSE ... ENDMATCH
*C*140	T_Ifmatch2_Assigns	T_Statement	*B*2, *B*7, *B*7	IFMATCH2 Assigns ... THEN ... ELSE ... ENDMATCH
*C*141	T_Ifmatch2_Cond	T_Statement	*A*3, *B*7, *B*7	IFMATCH2 Condition ... THEN ... ELSE ... ENDMATCH
*C*142	T_Ifmatch2_Defn	T_Statement	*A*4, *B*7, *B*7	IFMATCH2 Definition ... THEN ... ELSE ... ENDMATCH
*C*143	T_Ifmatch2_Defns	T_Statement	*B*3, *B*7, *B*7	IFMATCH2 Definitions ... THEN ... ELSE ... ENDMATCH

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*144	T_Ifmatch2_Expn	T_Statement	*A*5, *B*7, *B*7	IFMATCH2 Expression ... THEN ... ELSE ... ENDMATCH
*C*145	T_Ifmatch2_Expns	T_Statement	*B*4, *B*7, *B*7	IFMATCH2 Expressions ... THEN ... ELSE ... ENDMATCH
*C*146	T_Ifmatch2_Guarded	T_Statement	*A*6, *B*7, *B*7	IFMATCH2 Guarded ... THEN ... ELSE ... ENDMATCH
*C*147	T_Ifmatch2_Lvalue	T_Statement	*A*7, *B*7, *B*7	IFMATCH2 Lvalue ... THEN ... ELSE ... ENDMATCH
*C*148	T_Ifmatch2_Lvalues	T_Statement	*B*6, *B*7, *B*7	IFMATCH2 Lvalues ... THEN ... ELSE ... ENDMATCH
*C*149	T_Ifmatch2_Stat	T_Statement	*A*9, *B*7, *B*7	IFMATCH2 Statement ... THEN ... ELSE ... ENDMATCH
*C*150	T_Ifmatch2_Stats	T_Statement	*B*7, *B*7, *B*7	IFMATCH2 Statements ... THEN ... ELSE ... ENDMATCH
*C*151	T_Ifmatch_Action	T_Statement	*A*1, *B*7, *B*7	IFMATCH Action ... THEN ... ELSE ... ENDMATCH
*C*152	T_Ifmatch_Assign	T_Statement	*A*2, *B*7, *B*7	IFMATCH Assign ... THEN ... ELSE ... ENDMATCH
*C*153	T_Ifmatch_Assigns	T_Statement	*B*2, *B*7, *B*7	IFMATCH Assigns ... THEN ... ELSE ... ENDMATCH
*C*154	T_Ifmatch_Cond	T_Statement	*A*3, *B*7, *B*7	IFMATCH Condition ... THEN ... ELSE ... ENDMATCH
*C*155	T_Ifmatch_Defn	T_Statement	*A*4, *B*7, *B*7	IFMATCH Definition ... THEN ... ELSE ... ENDMATCH
*C*156	T_Ifmatch_Defns	T_Statement	*B*3, *B*7, *B*7	IFMATCH Definitions ... THEN ... ELSE ... ENDMATCH
*C*157	T_Ifmatch_Expn	T_Statement	*A*5, *B*7, *B*7	IFMATCH Expression ... THEN ... ELSE ... ENDMATCH
*C*158	T_Ifmatch_Expns	T_Statement	*B*4, *B*7, *B*7	IFMATCH Expressions ... THEN ... ELSE ... ENDMATCH
*C*159	T_Ifmatch_Guarded	T_Statement	*A*6, *B*7, *B*7	IFMATCH Guarded ... THEN ... ELSE ... ENDMATCH
*C*160	T_Ifmatch_Lvalue	T_Statement	*A*7, *B*7, *B*7	IFMATCH Lvalue ... THEN ... ELSE ... ENDMATCH
*C*161	T_Ifmatch_Lvalues	T_Statement	*B*6, *B*7, *B*7	IFMATCH Lvalues ... THEN ... ELSE ... ENDMATCH

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*162	T_Ifmatch_Stat	T_Statement	*A*9, *B*7, *B*7	IFMATCH Statement ... THEN ... ELSE ... ENDMATCH
*C*163	T_Ifmatch_Stats	T_Statement	*B*7, *B*7, *B*7	IFMATCH Statements ... THEN ... ELSE ... ENDMATCH
*C*164	T_Implies	T_Condition	*A*3, *A*3	IMPLIES ?(..., ...)
*C*165	T_In	T_Condition	*A*5, *A*5	... IN ...
*C*166	T_Index	T_Expression	*B*4	INDEX (...)
*C*167	T_Int	T_Expression	*A*5	INT (...)
*C*168	T_Intersection	T_Expression	*A*5, *A*5	... /\ ...
*C*169	T_Invert	T_Expression	*A*5	
*C*170	T_Join	T_Statement	*B*7, *B*7	JOIN ..., ... ENDJOIN
*C*171	T_Last	T_Expression	*A*5	LAST (...)
*C*172	T_Length	T_Expression	*A*5	LENGTH (...)
*C*173	T_Less	T_Condition	*A*5, *A*5	... < ...
*C*174	T_Less_Eq	T_Condition	*A*5, *A*5	... <= ...
*C*175	T_Lvalue_Int_Any	T_Lvalue	*A*5	
*C*176	T_Lvalue_Int_One	T_Lvalue	*A*5	
*C*177	T_Lvalue_Pat_Any	T_Lvalue		* ...
*C*178	T_Lvalue_Pat_Many	T_Lvalue		+ ...
*C*179	T_Lvalue_Pat_One	T_Lvalue		? ...

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*180	T_Lvalue_Val_Any	T_Lvalue		
*C*181	T_Lvalue_Val_One	T_Lvalue		
*C*182	T_Map	T_Expression	*A*8, *A*5	MAP ("...", ...)
*C*183	T_Maphash	T_Statement	*A*8, *A*5	MAPHASH ("...", ...)
*C*184	T_Max	T_Expression	*A*5	MAX (...)
*C*185	T_Member	T_Condition	*A*5, *A*5	MEMBER ?(..., ...)
*C*186	T_Min	T_Expression	*A*5	MIN (...)
*C*187	T_Minus	T_Expression	*A*5, *A*5	... - ...
*C*188	T_Mod	T_Expression	*A*5, *A*5	... MOD ...
*C*189	T_MW_BFunct	T_Statement	*A*8, *B*6, *B*2, *B*7, *A*3	MW_BFUNCT ... ?(...) == VAR <...>: ...; (...) END
*C*190	T_MW_BFunct_Call	T_Condition	*A*8, *B*4	... ?(...)
*C*191	T_MW_Funct	T_Statement	*A*8, *B*6, *B*2, *B*7, *A*5	MW_FUNCT ... (...) == VAR <...>: ...; (...) END
*C*192	T_MW_Funct_Call	T_Expression	*A*8, *B*4	... (...)
*C*193	T_MW_Proc	T_Statement	*A*8, *B*6, *B*6, *B*7	MW_PROC ... (... VAR ...) == ... END
*C*194	T_MW_Proc_Call	T_Statement	*A*8, *B*4, *B*6	... (...)
*C*195	T_Name_Int_One	T_Name	*A*5	
*C*196	T_Name_Pat_One	T_Name		? ...
*C*197	T_Name_Val_One	T_Name		

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*198	T_Negate	T_Expression	*A*5	- ...
*C*199	T_Not	T_Condition	*A*3	NOT ...
*C*200	T_Not_Equal	T_Condition	*A*5, *A*5	... <> ...
*C*201	T_Not_In	T_Condition	*A*5, *A*5	... NOTIN ...
*C*202	T_Number	T_Expression		...
*C*203	T_Numberq	T_Condition	*A*5	
*C*204	T_Odd	T_Condition	*A*5	ODD ?(...)
*C*205	T_Or	T_Condition	*A*3, *A*3	... OR ...
*C*206	T_Plus	T_Expression	*A*5, *A*5	... + ...
*C*207	T_Pop	T_Statement	*A*7, *A*7	POP (..., ...)
*C*208	T_Powerset	T_Expression	*A*5	POWERSET (...)
*C*209	T_Primed_Var	T_Expression		
*C*210	T_Print	T_Statement	*B*4	PRINT (...)
*C*211	T_Printflush	T_Statement	*B*4	PRINTFLUSH (...)
*C*212	T_Proc	T_Definition	*A*8, *B*6, *B*6, *B*7	PROC ... (... VAR ...) == ... END
*C*213	T_Proc_Call	T_Statement	*A*8, *B*4, *B*6	... (... VAR ...)
*C*214	T_Push	T_Statement	*A*7, *A*5	PUSH (..., ...)
*C*215	T_Put_Hash	T_Statement	*A*7, *A*5, *A*5	... .(...) := ...

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*216	T_Reduce	T_Expression	*A*8, *A*5	REDUCE ("...", ...)
*C*217	T_Rel_Seg	T_Expression	*A*5, *A*5, *A*5	... [..., ...]
*C*218	T_Rel_Seg_Lvalue	T_Lvalue	*A*7, *A*5, *A*5	... (..., ...)
*C*219	T_Reverse	T_Expression	*A*5	REVERSE (...)
*C*220	T_Sequence	T_Expression	*B*4	<...>
*C*221	T_Sequenceq	T_Condition	*A*5	
*C*222	T_Set	T_Expression	*A*5, *A*3	{... | ...}
*C*223	T_Set_Diff	T_Expression	*A*5, *A*5	... \ ...
*C*224	T_Sgn	T_Expression	*A*5	SGN (...)
*C*225	T_Skip	T_Statement		SKIP
*C*226	T_Slength	T_Expression	*A*5	SLENGTH (...)
*C*227	T_Spec	T_Statement	*B*6, *A*3	SPEC <...>: ... ENDSPEC
*C*228	T_Stat_Int_Any	T_Statement	*A*5	
*C*229	T_Stat_Int_One	T_Statement	*A*5	
*C*230	T_Stat_Pat_Any	T_Statement		* ...
*C*231	T_Stat_Pat_Many	T_Statement		+ ...
*C*232	T_Stat_Pat_One	T_Statement		? ...
*C*233	T_Stat_Place	T_Statement		$Statement$

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*234	T_Stat_Val_Any	T_Statement		
*C*235	T_Stat_Val_One	T_Statement		
*C*236	T_String	T_Expression		"..."
*C*237	T_Stringq	T_Condition	*A*5	
*C*238	T_Struct	T_Expression	*A*8, *A*5	... .
*C*239	T_Struct_Lvalue	T_Lvalue	*A*8, *A*7	... .
*C*240	T_Subset	T_Condition	*A*5, *A*5	SUBSET ?(..., ...)
*C*241	T_Substr	T_Expression	*B*4	SUBSTR (...)
*C*242	T_Sub_Seg	T_Expression	*A*5, *A*5, *A*5	... [... .. ...]
*C*243	T_Sub_Seg_Lvalue	T_Lvalue	*A*7, *A*5, *A*5	... (... .. ...)
*C*244	T_Tail	T_Expression	*A*5	TAIL (...)
*C*245	T_Times	T_Expression	*A*5, *A*5	... * ...
*C*246	T_True	T_Condition		TRUE
*C*247	T_Union	T_Expression	*A*5, *A*5	... \/ ...
*C*248	T_Var	T_Statement	*B*2, *B*7	VAR <...>: ... ENDVAR
*C*249	T_Variable	T_Expression		...
*C*250	T_Var_Lvalue	T_Lvalue		...
*C*251	T_Var_Place	T_Expression		Var

Continued on next page

Specific types in WSL - continued from previous page.

ID	Specific Type	General Type	Subtypes	WSL syntax
*C*252	T_Where	T_Statement	*B*7, *B*3	BEGIN ... WHERE ... END
*C*253	T_While	T_Statement	*A*3, *B*7	WHILE ... DO ... OD
*C*254	T_X_BFunct_Call	T_Condition	*A*8, *B*4	!XC ... ?(...)
*C*255	T_X_Funct_Call	T_Expression	*A*8, *B*4	!XF ... (...)
*C*256	T_X_Proc_Call	T_Statement	*A*8, *B*4	!XP ... (...)

Appendix B

Description of the FermaT Transformations

This chapter provides a description of the FermaT transformations which have been used in this thesis. The transformations are written in $\mathcal{META}$WSL and can only be applied on a particular AST type within the WSL code of a program. Some of them are only available in combination with the commercial version (FermaT 2) of the FermaT Transformation Engine (FTE) but most of them are also available in combination with the public version (FermaT 3). The FermaT transformations are split into several groups where it is possible that an individual transformation belongs to more than one group.

B.1 Abort Processing

The Abort Processing transformation uses *ABORT* statements to simplify the selected statements. The selected AST type to transform the WSL code of Listing B.1 into the WSL code of Listing B.2 is the specific type T_Floop but the Abort Processing can be applied on any statements which contain the specific type T_Abort. The transformation belongs to the group *Simplify* and is available in both versions of the FTE.

Listing B.1: Abort Processing Example: Initial Program P_0 WSL Code.

```
1 DO
2   IF  i  >=  10  THEN
3     EXIT(1)
4   FI;
```

```
5     i := i + 1;
6     ABORT
7  OD
```

Listing B.2: Abort Processing Example: Final Program P_n WSL Code.

```
1  DO
2     ABORT
3  OD
```

B.2 Absorb Left

The Absorb Left transformation absorbs the preceding statement into the selected statement. The selected AST type to transform the WSL code of Listing B.3 into the WSL code of Listing B.4 is the specific type T_For but the Absorb Left can be applied on any statement which has another one to the left. The transformation belongs to the group *Join* and is available in both versions of the FTE.

Listing B.3: Absorb Left Example: Initial Program P_0 WSL Code.

```
1  k := 1;
2  FOR i := 0 TO 10 STEP 2 DO
3     j := j + i;
4     SKIP
5  OD
```

Listing B.4: Absorb Left Example: Final Program P_n WSL Code.

```
1  FOR i := 0 TO 10 STEP 2 DO
2     k := 1;
3     j := j + i;
4     SKIP
5  OD
```

B.3 Add Assertion

The Add Assertion transformation adds an assertion after the selected statement if some suitable information can be ascertained. The selected AST type to transform the WSL code of Listing B.5 into the WSL code of Listing B.6 is the first T_Assign but the Add Assertion can be applied on any general type T_Assign and on the specific types T_Abort, T_Assert, T_Cond and T_While as well. The transformation belongs to the group *Insert* and is available in both versions of the FTE.

Listing B.5: Add Assertion Example: Initial Program P_0 WSL Code.

```
1 i := 0;
2 IF i = 0 THEN
3    j := 1
4 FI
```

Listing B.6: Add Assertion Example: Final Program P_n WSL Code.

```
1 i := 0;
2 {i = 0};
3 IF i = 0 THEN
4    j := 1
5 FI
```

B.4 Collapse Action System

The Collapse Action System transformation uses simplifications and substitutions to convert the selected action system into statements which are possibly inside a *DO* loop. The selected AST type to transform the WSL code of Listing B.7 into the WSL code of Listing B.8 is the specific type T_A_S. Furthermore, this is the only AST type on which the Collapse Action System can be applied. The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.7: Collapse Action System Example: Initial Program P_0 WSL Code.

```
1 ACTIONS PROG:
2    PROG ==
3       CALL A
```

```
4    END
5    A ==
6       IF  i  <>  j  THEN
7          i  :=  j ;
8          CALL A
9       ELSE
10         CALL B
11      FI
12   END
13   B ==
14      CALL Z
15   END
16 ENDACTIONS
```

Listing B.8: Collapse Action System Example: Final Program P_n WSL Code.

```
1 DO
2    IF  i  <>  j  THEN
3       i  :=  j
4    ELSE
5       EXIT(1)
6    FI
7 OD
```

B.5 Constant Propagation

The Constant Propagation transformation searches for assignments of constants to variables and propagates these constants through the selected statements by replacing variables with the appropriate value. Afterwards, it tries to simplify the statements which are affected by the replaced variables. The selected AST type to transform the WSL code of Listing B.9 into the WSL code of Listing B.10 is the first T_Statements but the Constant Propagation can be applied on any statements which contain the specific type T_Assign. The transformation belongs to the group *Simplify* and is available in both versions of the FTE.

Listing B.9: Constant Propagation Example: Initial Program P_0 WSL Code.

```
1 i := 5;
2 IF i = 5 THEN
3   j := 0
4 FI
```

Listing B.10: Constant Propagation Example: Final Program P_n WSL Code.

```
1 i := 5;
2 j := 0
```

B.6 Delete All Assertions

The Delete All Assertions deletes all assertions within the selected statements. The selected AST type to transform the WSL code of Listing B.11 into the WSL code of Listing B.12 is the first T_Statements but the Delete All Assertions can be applied on any statements which contain the specific type T_Assert. The transformation belongs to the group *Simplify* and is available in both versions of the FTE.

Listing B.11: Delete All Assertions Example: Initial Program P_0 WSL Code.

```
1 i := 5;
2 {i = 5};
3 IF i = 5 THEN
4   j := 0;
5   {j = 0}
6 FI
```

Listing B.12: Delete All Assertions Example: Final Program P_n WSL Code.

```
1 i := 5;
2 IF i = 5 THEN
3   j := 0
4 FI
```

B.7 Delete All Redundant

The Delete All Redundant transformation deletes all redundant statements within the selected statements. The selected AST type to transform the WSL code of Listing B.13 into the WSL code of Listing B.14 is the group type T_Statements but the Delete All Redundant can be applied on any statements. The transformation belongs to the group *Delete* and is available in both versions of the FTE.

Listing B.13: Delete All Redundant Example: Initial Program P_0 WSL Code.

```
1 i := 10;
2 j := 20;
3 i := i + j;
4 i := 15
```

Listing B.14: Delete All Redundant Example: Final Program P_n WSL Code.

```
1 j := 20;
2 i := 15
```

B.8 Delete All Skips

The Delete All Skips transformation deletes all *SKIP* statements within the selected statements. The selected AST type to transform the WSL code of Listing B.15 into the WSL code of Listing B.16 is the first T_Statements but the Delete All Skips can be applied on any statements which contain the specific type T_Skip. The transformation belongs to the group *Delete* and to the group *Simplify*. It is available in both versions of the FTE.

Listing B.15: Delete All Skips Example: Initial Program P_0 WSL Code.

```
1 i := 5;
2 SKIP;
3 IF i = 5 THEN
4   SKIP;
5   j := 0
6 FI
```

Listing B.16: Delete All Skips Example: Final Program P_n WSL Code.

```
1 i := 5;
2 IF i = 5 THEN
3   j := 0
4 FI
```

B.9 Delete Unreachable Code

The Delete Unreachable Code transformation deletes unreachable statements within the selected statements. It considers a statement as unreachable if there is no path in the control flow graph from the start node to the node which contains the selected statement. The selected AST type to transform the WSL code of Listing B.17 into the WSL code of Listing B.18 is the last T_Assign but the Delete Unreachable Code can be applied on any statements. The transformation belongs to the group *Simplify* and is available in both versions of the FTE.

Listing B.17: Delete Unreachable Code Example: Initial Program P_0 WSL Code.

```
1 DO
2   i := 0;
3   EXIT(1);
4   i := 1
5 OD
```

Listing B.18: Delete Unreachable Code Example: Final Program P_n WSL Code.

```
1 DO
2   i := 0;
3   EXIT(1)
4 OD
```

B.10 Dijkstra Do to Floop

The Dijkstra Do to Floop transformation converts the selected *D_DO* loop into a *DO* loop. The selected AST type to transform the WSL code of Listing B.19 into the WSL code of

Listing B.20 is the specific type T_D_DO. Furthermore, this is the only AST type on which the Dijkstra Do to Floop can be applied. The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.19: Dijkstra Do to Floop Example: Initial Program P_0 WSL Code.

```
1 D_DO EVEN?(i) AND i <> 0 ->
2    i := i / 2
3 [] ODD?(i) ->
4    i := i - 1
5 OD
```

Listing B.20: Dijkstra Do to Floop Example: Final Program P_n WSL Code.

```
1 DO
2    D_IF EVEN?(i) AND i <> 0 ->
3       i := i / 2
4    [] ODD?(i) ->
5       i := i - 1
6    [] i = 0 ->
7       EXIT(1)
8    FI
9 OD
```

B.11 Double to Single Loop

The Double to Single Loop transformation converts the selected double nested loop into a single loop if this is possible without significantly increasing the size of the program. The selected AST type to transform the WSL code of Listing B.21 into the WSL code of Listing B.22 is the first T_Floop. Furthermore, this specific type is the only AST type on which the Dijkstra Do to Floop can be applied. The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.21: Double to Single Loop Example: Initial Program P_0 WSL Code.

```
1 DO
2    DO
3       IF i = j THEN
```

```
4         EXIT(2)
5       FI;
6       i := j
7     OD
8   OD
```

Listing B.22: Double to Single Loop Example: Final Program P_n WSL Code.

```
1 DO
2   IF i = j THEN
3     EXIT(1)
4   FI;
5   i := j
6 OD
```

B.12 Else If to Elsif

The Else If to Elsif transformation converts the selected *ELSE* clause which contains an *IF* statement into an *ELSIF* clause. The selected AST type to transform the WSL code of Listing B.23 into the WSL code of Listing B.24 is the specific type T_Cond but the Else If to Elsif can be applied on the general type T_Guarded as well. The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.23: Else If to Elsif Example: Initial Program P_0 WSL Code.

```
1 IF i = 0 THEN
2   j := 10;
3   k := 20
4 ELSE
5   IF i = 1 THEN
6     j := 20;
7     k := 10
8   FI
9 FI
```

Listing B.24: Else If to Elsif Example: Final Program P_n WSL Code.

```
1 IF i = 0 THEN
```

```
2     j := 10;
3     k := 20
4  ELSIF i = 1 THEN
5     j := 20;
6     k := 10
7  FI
```

B.13 Fix Assembler

The Fix Assembler transformation restructures and simplifies WSL code which has been translated from assembler. Additionally, it removes features which have been introduced to WSL to keep the assembler to WSL translator as simple as possible. The selected AST type to transform the WSL code of Listing B.25 into the WSL code of Listing B.26 is the specific type T_Where but the Fix Assembler can be applied on any statements. The transformation belongs to the group *Simplify* and is available only in the commercial version (FermaT 2) of the FTE.

Listing B.25: Fix Assembler Example: Initial Program P_0 WSL Code.

```
1  BEGIN
2     i := 0;
3     j := 5;
4     IF i = j THEN
5        k := 1;
6        ABORT
7     ELSIF i > j THEN
8        k := 2;
9        A( VAR );
10       SKIP
11    ELSE
12       k := 3;
13       SKIP
14    FI;
15    WHILE j > 0 DO
16       k := k + i + j;
```

```
17      j := j - 1
18    OD;
19    i := k - 1;
20    SKIP
21  WHERE
22    PROC A( VAR ) ==
23      j := 2;
24      k := i + k;
25      l := j ** k
26    END
27  END
```

Listing B.26: Fix Assembler Example: Final Program P_n WSL Code.

```
1 j := 5;
2 k := 3;
3 WHILE j > 0 DO
4   k := j + k;
5   j := j - 1
6 OD;
7 i := k - 1
```

B.14 Fix Dispatch

The Fix Dispatch transformation converts a collection of actions with a single entry point and multiple exit points into a nested action system with a single exit point. Afterwards, the nested action system will be converted into a *WHERE* clause and the dispatch will be simplified via constant propagation. The selected AST type to transform the WSL code of Listing B.27 into the WSL code of Listing B.27 is the specific type T_A_S. Furthermore, this is the only AST type on which the Fix Dispatch can be applied. The transformation belongs to the group *Complex* and to the group *Rewrite*. It is available in both versions of the FTE.

Listing B.27: Fix Dispatch Example: Initial Program P_0 WSL Code.

```
1 ACTIONS PROG:
```

```
  PROG ==
    IF i = 1 THEN
      CALL A
    FI;
    r12 := 3644;
    CALL A
  END
  A ==
    destination := r12;
    CALL dispatch
  END
  B ==
    destination := 2526;
    CALL dispatch
  END
  dispatch ==
    IF destination = 2526 THEN
      CALL PROG
    ELSIF destination = 3560 THEN
      CALL A
    ELSIF destination = 3644 THEN
      CALL B
    ELSE
      CALL Z
    FI
  END
ENDACTIONS
```

Listing B.28: Fix Dispatch Example: Final Program P_n WSL Code.

```
BEGIN
  ACTIONS PROG:
    PROG ==
      IF i = 1 THEN
        A( VAR );
```

```
6           CALL dispatch
7         FI;
8         r12 := NOTUSED_3644;
9         A( VAR );
10        CALL B
11      END
12      B ==
13        destination := 2526;
14        CALL dispatch
15      END
16      dispatch ==
17        IF destination = 2526 THEN
18          CALL PROG
19        ELSIF destination = 3560 THEN
20          A( VAR );
21          CALL dispatch
22        ELSIF destination = 3644 THEN
23          CALL B
24        ELSE
25          CALL Z
26        FI
27      END
28    ENDACTIONS
29  WHERE
30    PROC A( VAR ) ==
31      destination := r12;
32      exit_flag := 0
33    END
34  END
```

B.15 Fix Init

The Fix Init transformation deletes assertions, comments, register initialisations and stack operations within the selected statements. The selected AST type to transform the WSL code of Listing B.29 into the WSL code of Listing B.30 is the group type T_A_S but the Fix Init can be applied on any statements. The transformation belongs to the group *Simplify* and is available only in the commercial version (FermaT 2) of the FTE.

Listing B.29: Fix Init Example: Initial Program P_0 WSL Code.

```
1  ACTIONS PROG:
2    PROG ==
3      C:"first comment";
4      r1 := __r1_init__;
5      r3 := __r3_init__;
6      r7 := __r7_init__;
7      r8 := __r8_init__;
8      r9 := __r9_init__;
9      r10 := __r10_init__;
10     r11 := __r11_init__;
11     r12 := __r12_init__;
12     r13 := __r13_init__;
13     r14 := __r14_init__;
14     CALL A
15   END
16   A ==
17     C:"second comment";
18     IF 3 = 0 THEN
19       i := r1
20     ELSE
21       i := r3;
22       r3 := 0
23     FI;
24     CALL B
25   END
26   B ==
```

```
27      C:"third comment";
28      r15 := i;
29      CALL Z
30    END
31  ENDACTIONS
```

Listing B.30: Fix Init Example: Final Program P_n WSL Code.

```
1  ACTIONS PROG:
2    PROG ==
3      CALL A
4    END
5    A ==
6      i := r3;
7      r3 := 0;
8      CALL B
9    END
10   B ==
11     r15 := i;
12     CALL Z
13   END
14 ENDACTIONS
```

B.16 Floop to While

The Floop to While transformation converts the selected *DO* loop into a *WHILE* loop. The selected AST type to transform the WSL code of Listing B.31 into the WSL code of Listing B.32 is the specific type T_Floop. Furthermore, this is the only AST type on which the Floop to While can be applied. The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.31: Floop to While Example: Initial Program P_0 WSL Code.

```
1 i := 0;
2 DO
3   IF FALSE THEN
```

```
4       EXIT(1)
5     FI;
6     i := i + 1
7  OD;
8  j := i
```

Listing B.32: Floop to While Example: Final Program P_n WSL Code.

```
1  i := 0;
2  WHILE TRUE DO
3     i := i + 1
4  OD;
5  j := i
```

B.17 For to While

The For to While transformation converts the selected *FOR* loop into a *WHILE* loop. The selected AST type to transform the WSL code of Listing B.33 into the WSL code of Listing B.34 is the specific type T_Floop. Furthermore, this is the only AST type on which the For to While can be applied. The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.33: For to While Example: Initial Program P_0 WSL Code.

```
1  j := 0;
2  FOR i := 0 TO 10 STEP 1 DO
3     j := j + i
4  OD
```

Listing B.34: For to While Example: Final Program P_n WSL Code.

```
1  j := 0;
2  VAR <i := 0>:
3     WHILE i <= 10 DO
4        j := j + i;
5        i := i + 1
6     OD
7  ENDVAR
```

B.18 Insert Assertions

The Insert Assertions transformation inserts assertions into the selected statement if some suitable information can be ascertained. The selected AST type to transform the WSL code of Listing B.35 into the WSL code of Listing B.36 is the specific type T_Cond but the Insert Assertions can be applied on the general type T_Assign and on the specific types T_Abort, T_Assert and T_While as well. The transformation belongs to the group *Insert* and is available in both versions of the FTE.

Listing B.35: Insert Assertions Example: Initial Program P_0 WSL Code.

```
1  IF i < 0 THEN
2     k := 0
3  ELSIF i > 0 AND i < 10 THEN
4     k := 1
5  ELSIF i <> 0 AND i <> 10 THEN
6     k := 2
7  ELSE
8     k := 3
9  FI
```

Listing B.36: Insert Assertions Example: Final Program P_n WSL Code.

```
1  IF i < 0 THEN
2     {i < 0};
3     k := 0
4  ELSIF i > 0 AND i < 10 THEN
5     {i > 0 AND i < 10};
6     k := 1
7  ELSIF i <> 0 AND i <> 10 THEN
8     {i <> 0 AND i <> 10};
9     k := 2
10 ELSE
11    {i = 0 OR i = 10};
12    k := 3
13 FI
```

B.19 Loop Doubling

The Loop Doubling transformation duplicates the body of the selected loop. The selected AST type to transform the WSL code of Listing B.37 into the WSL code of Listing B.38 is the specific type T_Floop but the Loop Doubling can be applied on the specific types T_For and T_While as well. The transformation belongs to the group *Insert* and is available in both versions of the FTE.

Listing B.37: Loop Doubling Example: Initial Program P_0 WSL Code.

```
1 DO
2   IF i = 0 THEN
3     j := 0;
4     EXIT(1)
5   ELSE
6     j := 1;
7     EXIT(1)
8   FI
9 OD
```

Listing B.38: Loop Doubling Example: Final Program P_n WSL Code.

```
1 DO
2   IF i = 0 THEN
3     j := 0;
4     EXIT(1)
5   ELSE
6     j := 1;
7     EXIT(1)
8   FI;
9   IF i = 0 THEN
10    j := 0;
11    EXIT(1)
12  ELSE
13    j := 1;
14    EXIT(1)
15  FI
```

```
16 OD
```

B.20 Merge Left

The Merge Left transformation merges the selected statement into the preceding statement. The selected AST type to transform the WSL code of Listing B.39 into the WSL code of Listing B.40 is the last T_Assign but the Merge Left can be applied on any statement which has another one to the left. The transformation belongs to the group *Join* and is available in both versions of the FTE.

Listing B.39: Merge Left Example: Initial Program P_0 WSL Code.

```
1 j := 0;
2 k := j;
3 k := k + 1
```

Listing B.40: Merge Left Example: Final Program P_n WSL Code.

```
1 j := 0;
2 k := j + 1
```

B.21 Merge Right

The Merge Right transformation merges the selected statement into the following statement. The selected AST type to transform the WSL code of Listing B.41 into the WSL code of Listing B.42 is the second T_Assign but the Merge Right can be applied on any statement which has another one to the right. The transformation belongs to the group *Join* and is available in both versions of the FTE.

Listing B.41: Merge Right Example: Initial Program P_0 WSL Code.

```
1 j := 0;
2 k := j;
3 k := k + 1
```

Listing B.42: Merge Right Example: Final Program P_n WSL Code.

```
1 j := 0;
2 k := j + 1
```

B.22 Program to Specification

The Program to Specification transformation converts the selected program statements into semantically equivalent specification statements. The selected AST type to transform the WSL code of Listing B.43 into the WSL code of Listing B.44 is the first T_Statements but the Program to Specification can be applied on any statements which do not contain some kind of loop. The transformation belongs to the group *Abstraction* and is available in both versions of the FTE.

Listing B.43: Program to Specification Example: Initial Program P_0 WSL Code.

```
1 IF i = 1 THEN
2   j := 1
3 ELSE
4   j := 2
5 FI
```

Listing B.44: Program to Specification Example: Final Program P_n WSL Code.

```
1 SPEC <j>:
2   j' = 1 AND i = 1 OR j' = 2 AND i <> 1
3 ENDSPEC
```

B.23 Prune Dispatch

The Prune Dispatch transformation deletes *GUARDED* statements within a dispatch action which test variables against never assigned constants. The selected AST type to transform the WSL code of Listing B.45 into the WSL code of Listing B.46 is the specific type T_A_S but the Prune Dispatch can be applied on any action with the name "dispatch" as well. The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.45: Prune Dispatch Example: Initial Program P_0 WSL Code.

```
1 ACTIONS PROG:
2   PROG ==
3     IF i = 0 THEN
4       destination := 2512
```

```
5       ELSE
6         destination := 2530
7       FI;
8       CALL A
9     END
10    A ==
11      i := 0;
12      CALL dispatch
13    END
14    dispatch ==
15      IF destination = 2512 THEN
16        i := 0;
17        j := 0;
18        CALL Z
19      ELSIF destination = 2522 THEN
20        i := i + 1;
21        j := 0;
22        CALL Z
23      ELSIF destination = 2526 THEN
24        i := 0;
25        j := j + 1;
26        CALL Z
27      ELSIF destination = 2530 THEN
28        i := i + 1;
29        j := j + 1;
30        CALL Z
31      FI
32    END
33  ENDACTIONS
```

Listing B.46: Prune Dispatch Example: Final Program P_n WSL Code.

```
1   ACTIONS PROG:
2     PROG ==
3       IF i = 0 THEN
```

```
4         destination := 2512
5       ELSE
6         destination := 2530
7       FI;
8       CALL A
9     END
10    A ==
11      i := 0;
12      CALL dispatch
13    END
14    dispatch ==
15      IF destination = 2512 THEN
16        i := 0;
17        j := 0;
18        CALL Z
19      ELSIF destination = 2530 THEN
20        i := i + 1;
21        j := j + 1;
22        CALL Z
23      FI
24    END
25  ENDACTIONS
```

B.24 Reduce Nots

The Reduce Nots transformation reduces the number of negations within the conditions of the selected *IF* statement by switching the *GUARDED* statements. The selected AST type to transform the WSL code of Listing B.47 into the WSL code of Listing B.48 is the specific type T_Cond. Furthermore, this is the only AST type on which the Reduce Nots can be applied. The transformation belongs to the group *Rewrite* and is available only in the commercial version (FermaT 2) of the FTE.

Listing B.47: Reduce Nots Example: Initial Program P_0 WSL Code.

```
1 IF NOT i > j THEN
```

```
2    i := i + 1
3 FI
```

Listing B.48: Reduce Nots Example: Final Program P_n WSL Code.

```
1 IF i > j THEN
2    SKIP
3 ELSE
4    i := i + 1
5 FI
```

B.25 Refine Specification

The Refine Specification transformation converts the selected specification statement into a semantically equivalent program statement. The selected AST type to transform the WSL code of Listing B.49 into the WSL code of Listing B.50 is the specific type T_Spec. Furthermore, this is the only AST type on which the Refine Specification can be applied. The transformation belongs to the group *Refinement* and is available in both versions of the FTE.

Listing B.49: Refine Specification Example: Initial Program P_0 WSL Code.

```
1 SPEC <j>:
2    j' = 1 AND i = 1 OR j' = 2 AND i <> 1
3 ENDSPEC
```

Listing B.50: Refine Specification Example: Final Program P_n WSL Code.

```
1 IF i = 1 THEN
2    j := 1
3 ELSE
4    j := 2
5 FI
```

B.26 Remove All Redundant Variables

The Remove All Redundant Variables transformation removes all redundant variables within the selected statements. The selected AST type to transform the WSL code of Listing B.51 into the WSL code of Listing B.52 is the first T_Statements but the Remove All Redundant Variables can be applied on any statements which contain the specific type T_Var. The transformation belongs to the group *Delete* and is available in both versions of the FTE.

Listing B.51: Remove All Redundant Variables Example: Initial Program P_0 WSL Code.

```
1 VAR <i := 1, j := 2>:
2   WHILE i < 10 DO
3     i := i + j
4   OD
5 ENDVAR;
6 VAR <i := 2, j := 1>:
7   WHILE i < 8 DO
8     i := i + j
9   OD
10 ENDVAR
```

Listing B.52: Remove All Redundant Variables Example: Final Program P_n WSL Code.

```
1 VAR <i := 1>:
2   WHILE i < 10 DO
3     i := i + 2
4   OD
5 ENDVAR;
6 VAR <i := 2>:
7   WHILE i < 8 DO
8     i := i + 1
9   OD
10 ENDVAR
```

B.27 Remove Recursion in Action

The Remove Recursion in Action transformation replaces the body of the selected recursive action with a *DO* loop. The selected AST type to transform the WSL code of Listing B.53 into the WSL code of Listing B.54 is the general type T_Action. Furthermore, this is the only AST type on which the Remove Recursion in Action can be applied. The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.53: Remove Recursion in Action Example: Initial Program P_0 WSL Code.

```
1  ACTIONS PROG:
2    PROG ==
3      CALL A
4    END
5    A ==
6      IF i = j THEN
7        CALL B
8      FI;
9      i := i + 1;
10     CALL A
11   END
12   B ==
13     CALL Z
14   END
15 ENDACTIONS
```

Listing B.54: Remove Recursion in Action Example: Final Program P_n WSL Code.

```
1  ACTIONS PROG:
2    PROG ==
3      CALL A
4    END
5    A ==
6      DO
7        IF i = j THEN
8          EXIT(1)
9        FI;
```

```
10          i := i + 1;
11          SKIP
12        OD;
13        CALL B
14      END
15      B ==
16        CALL Z
17      END
18  ENDACTIONS
```

B.28 Rename Local Variables

The Rename Local Variables transformation removes the selected local *VAR* statement by renaming the variables. The selected AST type to transform the WSL code of Listing B.55 into the WSL code of Listing B.56 is the specific type T_Var. Furthermore, this is the only AST type on which the Rename Local Variables can be applied. The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.55: Rename Local Variables Example: Initial Program P_0 WSL Code.

```
1 VAR <i := 0, j := 3, k := 99>:
2   WHILE i < 10 DO
3     k := k - j - i
4   OD
5 ENDVAR
```

Listing B.56: Rename Local Variables Example: Final Program P_n WSL Code.

```
1 var_1__i := 0;
2 var_1__j := 3;
3 var_1__k := 99;
4 WHILE var_1__i < 10 DO
5   var_1__k := var_1__k - var_1__j - var_1__i
6 OD
```

B.29 Rename Procedure

The Rename Procedure transformation renames the selected procedure. The selected AST type to transform the WSL code of Listing B.57 into the WSL code of Listing B.58 is the last T_Proc. Furthermore, this is the only AST type on which the Rename Procedure can be applied. The used parameter and therefore the new procedure name is "B". The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.57: Rename Procedure Example: Initial Program P_0 WSL Code.

```
1  BEGIN
2    BEGIN
3      DO
4        IF i > 10 AND j > 10 THEN
5          A( VAR )
6        FI
7      OD
8    WHERE
9      PROC A( VAR ) ==
10       i := i + 1
11     END
12   END;
13   A( VAR )
14 WHERE
15   PROC A( VAR ) ==
16     j := j + 1
17   END
18 END
```

Listing B.58: Rename Procedure Example: Final Program P_n WSL Code.

```
1  BEGIN
2    BEGIN
3      DO
4        IF i > 10 AND j > 10 THEN
5          A( VAR )
6        FI
```

```
7       OD
8     WHERE
9       PROC A( VAR ) ==
10        i := i + 1
11      END
12    END;
13    B( VAR )
14  WHERE
15    PROC B( VAR ) ==
16      j := j + 1
17    END
18  END
```

B.30 Reverse Order

The Reverse Order transformation reverses the order of the selected condition or expression. The selected AST type to transform the WSL code of Listing B.59 into the WSL code of Listing B.60 is the specific type T_Cond but the Reverse Order can be applied on the specific type T_D_If as well. The transformation belongs to the group *Reorder* and is available in both versions of the FTE.

Listing B.59: Reverse Order Example: Initial Program P_0 WSL Code.

```
1 IF i > 0 THEN
2   j := i + 1;
3 ELSE
4   j := i - 1;
5 FI
```

Listing B.60: Reverse Order Example: Final Program P_n WSL Code.

```
1 IF i <= 0 THEN
2   j := i - 1
3 ELSE
4   j := i + 1
5 FI
```

B.31 Simplify

The Simplify transformation simplifies each item within the selected statements as far as possible. The selected AST type to transform the WSL code of Listing B.61 into the WSL code of Listing B.62 is the specific type T_Cond but the Simplify can be applied on any statements. The transformation belongs to the group *Simplify* and is available in both versions of the FTE.

Listing B.61: Simplify Example: Initial Program P_0 WSL Code.

```
1  IF  i > 10 THEN
2    DO
3       j := j + 1
4    OD;
5    j := 1;
6    k := 5
7  ELSIF  i > 20 THEN
8    j := 1 + 1;
9    k := 5
10 ELSIF  i <= 10 THEN
11   j := 1 + 1 + 1;
12   k := 5
13 ELSE
14   j := 1 + 1 + 1 + 1;
15   k := 5
16 FI
```

Listing B.62: Simplify Example: Final Program P_n WSL Code.

```
1  IF  i > 10 THEN
2    ABORT;
3    j := 1;
4    k := 5
5  ELSIF  i > 20 THEN
6    j := 2;
7    k := 5
8  ELSIF  i <= 10 THEN
```

```
    j := 3;
    k := 5
ELSE
    j := 4;
    k := 5
FI
```

B.32 Simplify Action System

The Simplify Action System transformation reduces the number of actions within the selected action system if this is possible without significantly increasing the size of the program. The selected AST type to transform the WSL code of Listing B.63 into the WSL code of Listing B.64 is the specific type T_A_S. Furthermore, this is the only AST type on which the Simplify Action System can be applied. The transformation belongs to the group *Simplify* and is available in both versions of the FTE.

Listing B.63: Simplify Action System Example: Initial Program P_0 WSL Code.

```
VAR <i := 0, j := 0>:
  ACTIONS PROG:
    PROG ==
      IF i = k THEN
        k := k + 1;
        CALL Z
      FI;
      i := i + 1;
      CALL A
    END
    A ==
      j := j + 1;
      k := j + a[i];
      CALL B
    END
    B ==
      IF i = 1 THEN
```

```
18            k := j + a[i]
19          FI;
20          j := 1;
21          CALL Z
22        END
23    ENDACTIONS
24 ENDVAR
```

Listing B.64: Simplify Action System Example: Final Program P_n WSL Code.

```
1  VAR <i := 0, j := 0>:
2    ACTIONS PROG:
3      PROG ==
4        IF i = k THEN
5          k := k + 1;
6          CALL Z
7        FI;
8        i := i + 1;
9        j := j + 1;
10       k := j + a[i];
11       IF i = 1 THEN
12         k := j + a[i]
13       FI;
14       j := 1;
15       CALL Z
16     END
17   ENDACTIONS
18 ENDVAR
```

B.33 Simplify If

The Simplify If transformation takes repeated statements out of the selected *IF* statement and simplifies the conditions as far as possible. Additionally, it removes any cases whose conditions imply earlier conditions and *FALSE* cases. The selected AST type to transform the WSL code of Listing B.65 into the WSL code of Listing B.66 is the specific type

T_Cond. Furthermore, this is the only AST type on which the Simplify If can be applied. The transformation belongs to the group *Simplify* and is available in both versions of the FTE.

Listing B.65: Simplify If Example: Initial Program P_0 WSL Code.

```
1  IF i > 10 THEN
2     j := 1;
3     k := 5
4  ELSIF i > 20 THEN
5     j := 2;
6     k := 5
7  ELSIF i <= 10 THEN
8     j := 3;
9     k := 5
10 ELSE
11    j := 4;
12    k := 5
13 FI
```

Listing B.66: Simplify If Example: Final Program P_n WSL Code.

```
1  IF i > 10 THEN
2     j := 1
3  ELSE
4     j := 3
5  FI;
6  k := 5
```

B.34 Simplify Item

The Simplify Item transformation simplifies the selected item as far as possible. The selected AST type to transform the WSL code of Listing B.67 into the WSL code of Listing B.68 is the general type T_Assign but the Simplify Item can be applied on the general types T_Expression and T_Guarded and on the specific types T_A_S, T_Cond, T_D_If, T_Floop, T_Var, T_Where and T_While as well. The transformation belongs to

the group *Simplify* and is available in both versions of the FTE.

Listing B.67: Simplify Item Example: Initial Program P_0 WSL Code.

```
1 i := j + j + k + j + k + k
```

Listing B.68: Simplify Item Example: Final Program P_n WSL Code.

```
1 i := 3 * (j + k)
```

B.35 Substitute and Delete

The Substitute and Delete transformation replaces all calls to the selected action, function or procedure with its definition if it is no recursion. Afterwards, it deletes the definition. The selected AST type to transform the WSL code of Listing B.69 into the WSL code of Listing B.70 is the general type T_Action but the Simplify Item can be applied on the specific types T_Funct and T_Proc as well. The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.69: Substitute and Delete Example: Initial Program P_0 WSL Code.

```
1  ACTIONS A:
2    A ==
3      i := i + 1;
4      CALL B
5    END
6    B ==
7      j := j + 1;
8      CALL C;
9      CALL D
10   END
11   C ==
12     CALL D
13   END
14   D ==
15     CALL A
16   END
17 ENDACTIONS
```

Listing B.70: Substitute and Delete Example: Final Program P_n WSL Code.

```
ACTIONS A:
  A ==
    i := i + 1;
    j := j + 1;
    CALL C;
    CALL D
  END
  C ==
    CALL D
  END
  D ==
    CALL A
  END
ENDACTIONS
```

B.36 Take Out Left

The Take Out Left transformation takes the selected statement out to the left of the enclosing structure. The selected AST type to transform the WSL code of Listing B.71 into the WSL code of Listing B.72 is the first T_Assign but the Take Out Left can be applied on any statements. The transformation belongs to the group *Move* and is available in both versions of the FTE.

Listing B.71: Take Out Left Example: Initial Program P_0 WSL Code.

```
IF i = 0 THEN
  j := 10;
  k := 0
ELSIF i > 0 THEN
  j := 10;
  k := 1
ELSE
  j := 10;
  k := 2
```

```
10 FI
```

Listing B.72: Take Out Left Example: Final Program P_n WSL Code.

```
1 j := 10;
2 IF i = 0 THEN
3    k := 0
4 ELSIF i > 0 THEN
5    k := 1
6 ELSE
7    k := 2
8 FI
```

B.37 Unroll Loop

The Unroll Loop transformation unrolls the first step of the selected loop by introducing an *IF* statement. The selected AST type to transform the WSL code of Listing B.73 into the WSL code of Listing B.74 is the specific type T_While but the Unroll Loop can be applied on the specific type T_Floop as well. The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.73: Unroll Loop Example: Initial Program P_0 WSL Code.

```
1 WHILE i > 0 DO
2    i := i - 1
3 OD
```

Listing B.74: Unroll Loop Example: Final Program P_n WSL Code.

```
1 IF i > 0 THEN
2    i := i - 1;
3    WHILE i > 0 DO
4       i := i - 1
5    OD
6 FI
```

B.38 Use Assertion

The Use Assertion transformation uses the selected assertion to simplify the following statements. The selected AST type to transform the WSL code of Listing B.75 into the WSL code of Listing B.76 is the specific type T_Assert. Furthermore, this is the only AST type on which the Use Assertion can be applied. The transformation belongs to the group *Apply or Use* and to the group *Simplify*. It is available in both versions of the FTE.

Listing B.75: Use Assertion Example: Initial Program P_0 WSL Code.

```
1 j := 0;
2 {i = 10};
3 WHILE i < 10 DO
4    j := j + i;
5    i := i + 1
6 OD
```

Listing B.76: Use Assertion Example: Final Program P_n WSL Code.

```
1 j := 0;
2 {i = 10};
3 SKIP
```

B.39 While to Floop

The While to Floop transformation converts a *WHILE* loop into a *DO* loop. The selected AST type to transform the WSL code of Listing B.77 into the WSL code of Listing B.78 is the specific type T_While. Furthermore, this is the only AST type on which the Collapse Action System can be applied. The transformation belongs to the group *Rewrite* and is available in both versions of the FTE.

Listing B.77: While to Floop Example: Initial Program P_0 WSL Code.

```
1 i := 0;
2 WHILE i < 10 DO
3    i := i + 1
4 OD;
5 j := i
```

Listing B.78: While to Floop Example: Final Program P_n WSL Code.

```
1 i := 0;
2 DO
3    IF i >= 10 THEN
4       EXIT(1)
5    FI;
6    i := i + 1
7 OD;
8 j := i
```

Appendix C

Case Study 2 WSL Code

Listing C.1: Case Study 2: Initial Program P_0 WSL Code.

```
1  VAR <destination := 0,
2       i := 0,
3       j := 0,
4       k := 0,
5       l := 0,
6       m := 0,
7       n := 0>:
8    ACTIONS PROG:
9      PROG ==
10       {i = 0};
11       IF i = 5 THEN
12         {j = 0};
13         IF j > 10 THEN
14           {k = 0};
15           IF k < 7 THEN
16             l := 9;
17             m := 7;
18             DO
19               DO
20                 IF n > 10 THEN
21                   EXIT(2)
```

```
22                  FI;
23                  m := (i + j) * k;
24                  n := n + 1
25                OD
26              OD;
27              SKIP
28            FI
29          FI
30        FI;
31        IF 5 > 7 THEN
32          SKIP;
33          CALL Z
34        ELSE
35          CALL B
36        FI
37      END
38      A ==
39        DO
40          DO
41            DO
42              IF i >= 20 THEN
43                EXIT(2)
44              ELSE
45                IF j >= 50 THEN
46                  SKIP;
47                  EXIT(2)
48                FI
49              FI;
50              i := i + m;
51              j := j + i;
52              m := m + n
53            OD
54          OD;
55          DO
```

```
56              DO
57                 k := k + 1;
58                 EXIT(1)
59              OD;
60              EXIT(1)
61           OD;
62           EXIT(1)
63        OD;
64        CALL B
65     END
66     B ==
67        FOR i := 0 TO 99 STEP 1 DO
68           j := j - 2;
69           IF i > 251 THEN
70              SKIP;
71              ABORT;
72              DO
73                 SKIP;
74                 i := i - 1
75              OD
76           FI
77        OD;
78        CALL C
79     END
80     C ==
81        IF n > 250 THEN
82           CALL J
83        ELSE
84           IF n > 220 THEN
85              CALL I
86           ELSE
87              IF n > 190 THEN
88                 CALL H
89              ELSE
```

```
90              IF n > 160 THEN
91                CALL G
92              ELSE
93                IF n > 130 THEN
94                  CALL F
95                ELSE
96                  CALL D
97                FI
98              FI
99            FI
100         FI
101       FI;
102       CALL D
103     END
104     D ==
105       BEGIN
106         P1( VAR )
107       WHERE
108         PROC P1( VAR ) ==
109           IF k = 0 THEN
110             i := 1;
111             j := 1
112           FI;
113           P2( VAR )
114         END
115         PROC P2( VAR ) ==
116           k := 0;
117           WHILE i < 5 DO
118             i := i + 1;
119             j := j + i * 2
120           OD;
121           P3( VAR )
122         END
123         PROC P3( VAR ) ==
```

```
124             m := 77;
125             n := i + j + 2;
126             IF l < 25 THEN
127               CALL G
128             ELSIF l < 15 THEN
129               CALL F
130             ELSE
131               CALL E
132             FI
133           END
134         END
135       END
136       E ==
137         IF i > j OR j < k THEN
138           ABORT;
139           IF i > j THEN
140             i := 53;
141             n := 0;
142             CALL H
143           ELSE
144             i := 44;
145             n := 1;
146             CALL H
147           FI
148         ELSIF i = j OR j = k THEN
149           IF i = j THEN
150             i := 33;
151             n := 2;
152             CALL G
153           ELSE
154             i := 33;
155             n := 2;
156             CALL G
157           FI
```

```
158         ELSE
159           CALL F
160         FI;
161         DO
162           SKIP;
163           EXIT(1);
164           ABORT;
165           IF n <> j THEN
166             CALL Z
167           ELSE
168             CALL B
169           FI
170         OD
171       END
172       F ==
173         ACTIONS PROG2:
174           PROG2 ==
175             i := 0;
176             SKIP;
177             CALL AA
178           END
179           AA ==
180             IF l <> n THEN
181               i := i + 1;
182               CALL BB
183             ELSE
184               i := i – 1
185             FI;
186             CALL BB
187           END
188           BB ==
189             k := 11;
190             m := 23;
191             SKIP;
```

```
192             CALL CC
193           END
194           CC ==
195             l := 21;
196             CALL G
197           END
198       ENDACTIONS
199   END
200   G ==
201     {i <> j};
202     IF i > j THEN
203       IF i > k THEN
204         IF i > l THEN
205           IF i > m THEN
206             IF i > n THEN
207               CALL Z
208             ELSE
209               DO
210                 DO
211                   IF i > n THEN
212                     EXIT(2)
213                   FI;
214                   i := i + 1;
215                   l := l - 1
216                 OD
217               OD
218             FI
219           FI
220         ELSE
221           DO
222             j := i - 3;
223             EXIT(1)
224           OD
225         FI
```

```
226             ELSE
227               DO
228                 WHILE i > j DO
229                   j := j + 1;
230                   k := k - 1;
231                   SKIP
232                 OD;
233                 EXIT(1)
234               OD
235             FI
236           ELSE
237             CALL I
238           FI;
239           CALL H
240       END
241       H ==
242           SKIP;
243           i := i + 1;
244           n := m - 5;
245           IF i > n THEN
246             ABORT;
247             CALL I
248           ELSE
249             CALL Z
250           FI
251       END
252       I ==
253           BEGIN
254             P1( VAR )
255           WHERE
256             PROC P1( VAR ) ==
257               m := 77;
258               n := i + j + 2;
259               IF l < 25 THEN
```

```
260             SKIP;
261             CALL L
262           ELSIF l < 15 THEN
263             SKIP;
264             CALL K
265           ELSE
266             SKIP;
267             P2( VAR )
268           FI
269         END
270         PROC P2( VAR ) ==
271           IF k = 0 THEN
272             i := 1;
273             j := 1
274           FI;
275           P3( VAR )
276         END
277         PROC P3( VAR ) ==
278           k := 0;
279           WHILE i < 5 DO
280             i := i + 1;
281             j := j + i * 2
282           OD;
283           SKIP;
284           CALL J
285         END
286       END
287     END
288     J ==
289       IF i > 273 THEN
290         j := 0;
291       CALL Z
292       ELSIF i > 329 THEN
293         j := 1;
```

```
294           CALL Z
295         ELSIF i > 452 THEN
296           j := 2;
297           CALL Z
298         ELSIF i > 531 THEN
299           j := 3;
300           CALL Z
301         ELSIF i > 621 THEN
302           j := 4;
303           CALL Z
304         ELSIF i > 710 THEN
305           j := 5;
306           CALL Z
307         ELSIF i > 867 THEN
308           j := 6;
309           CALL Z
310         ELSIF i > 965 THEN
311           j := 7;
312           CALL Z
313         ELSIF i > 1121 THEN
314           j := 8;
315           CALL Z
316         ELSIF i > 1452 THEN
317           j := 9;
318           CALL Z
319         ELSIF i > 1762 THEN
320           j := 10;
321           CALL Z
322         ELSIF i > 2257 THEN
323           j := 11;
324           CALL Z
325         ELSIF i > 2521 THEN
326           j := 12;
327           CALL Z
```

```
328         ELSE
329           CALL K
330         FI
331       END
332       K ==
333         k := 0;
334         IF l < 137 THEN
335           k := k + l;
336           l := l + 1;
337           SKIP;
338           IF l < 137 THEN
339             k := k + l;
340             l := l + 1;
341             SKIP;
342             IF l < 137 THEN
343               k := k + l;
344               l := l + 1;
345               SKIP;
346               IF l < 137 THEN
347                 k := k + l;
348                 l := l + 1;
349                 SKIP;
350                 WHILE l < 137 DO
351                   k := k + l;
352                   l := l + 1;
353                   SKIP
354                 OD
355               FI
356             FI
357           FI
358         FI;
359         CALL L
360       END
361       L ==
```

```
362         {i = 169};
363         IF i = 169 THEN
364           m := 0;
365           SKIP;
366           CALL Z
367         ELSIF k = 278 THEN
368           m := 1;
369           SKIP;
370           CALL Z
371         ELSIF k = 395 THEN
372           m := 2;
373           SKIP;
374           CALL Z
375         ELSIF k = 631 THEN
376           m := 3;
377           SKIP;
378           CALL Z
379         ELSIF k = 861 THEN
380           m := 4;
381           SKIP;
382           CALL Z
383         ELSIF k = 912 THEN
384           m := 5;
385           SKIP;
386           CALL Z
387         ELSIF k = 1021 THEN
388           m := 6;
389           SKIP;
390           CALL Z
391         ELSIF k = 1199 THEN
392           m := 7;
393           SKIP;
394           CALL Z
395         ELSIF k = 1392 THEN
```

```
396           m := 8;
397           SKIP;
398           CALL Z
399         ELSIF k = 1536 THEN
400           m := 9;
401           SKIP;
402           CALL Z
403         ELSIF k = 1743 THEN
404           m := 10;
405           SKIP;
406           CALL Z
407         ELSIF k = 1964 THEN
408           m := 11;
409           SKIP;
410           CALL Z
411         ELSIF k = 2243 THEN
412           m := 12;
413           SKIP;
414           CALL Z
415         ELSE
416           SKIP;
417           CALL M
418         FI
419       END
420       M ==
421         DO
422           DO
423             DO
424               i := i + 3;
425               j := k + m;
426               k := 5237;
427               l := l ** 2;
428               SKIP;
429               SKIP;
```

```
430             EXIT(3)
431           OD
432         OD
433       OD;
434       CALL N
435     END
436     N ==
437       n := i + j * m + k * n;
438       D_DO EVEN?(i) AND i <> 0 ->
439         i := i / 2;
440         CALL O
441       [] ODD?(i) ->
442         i := i - 1;
443         CALL P
444       OD
445     END
446     O ==
447       IF i <> 0 OR j = 0 OR k = m OR m = n THEN
448         IF i <> 0 THEN
449           i := i + 1
450         ELSE
451           l := l + 1
452         FI;
453         IF j = 0 THEN
454           j := 2761
455         ELSE
456           l := l + 2761
457         FI;
458         IF k = m THEN
459           k := k - 1 * l
460         ELSE
461           l := l + k
462         FI;
463         IF m = n THEN
```

```
464             m := m ** 2
465           ELSE
466             l := l + m ** 2
467           FI
468         FI;
469         CALL P
470       END
471       P ==
472         k := 0;
473         SKIP;
474         WHILE i > j DO
475           SKIP;
476           WHILE j > k DO
477             m := m + i;
478             k := k + 1
479           OD;
480           i := i - 1
481         OD;
482         CALL R
483       END
484       Q ==
485         VAR <o := 0>:
486           BEGIN
487             BEGIN
488               P1( VAR );
489               P2( VAR )
490             WHERE
491               PROC P1( VAR ) ==
492                 destination := 10698;
493                 IF i <> j THEN
494                   o := i;
495                   i := j;
496                   j := o
497                 FI
```

```
498             END
499             PROC P2( VAR ) ==
500               IF m <> n THEN
501                 o := m;
502                 m := n;
503                 n := o
504               FI
505             END
506           END;
507           P1( VAR );
508           P2( VAR );
509           P3( VAR );
510           P4( VAR )
511         WHERE
512           PROC P1( VAR ) ==
513             IF i > m THEN
514               i := i - m
515             ELSE
516               m := m - i
517             FI
518           END
519           PROC P2( VAR ) ==
520             i := i - 1;
521             j := j - 1;
522             k := k - 1;
523             l := l - 1;
524             m := m - 1;
525             n := n - 1
526           END
527           PROC P3( VAR ) ==
528             IF i = 3674 THEN
529               i := 674;
530               o := 1
531             ELSIF j = 7183 THEN
```

```
532                 j := 674;
533                 o := 1
534               ELSIF k = 7183 THEN
535                 k := 674;
536                 o := 1
537               ELSIF l = 7183 THEN
538                 l := 674;
539                 o := 1
540               ELSIF m = 7183 THEN
541                 m := 674;
542                 o := 1
543               ELSIF n = 7183 THEN
544                 n := 674;
545                 o := 1
546               ELSE
547                 o := 0
548               FI
549             END
550             PROC P4( VAR ) ==
551               IF o = 1 THEN
552                 CALL dispatch
553               ELSE
554                 CALL R
555               FI
556             END
557           END
558         ENDVAR
559       END
560       R ==
561         DO
562           SKIP;
563           DO
564             IF i > 1000 THEN
565               EXIT(2)
```

```
566             FI;
567             i := i + 1;
568             k := k + i;
569             n := n - i
570           OD
571         OD;
572         CALL S
573       END
574       S ==
575         {i < 2301};
576         IF i = 1352 THEN
577           j := 0;
578           CALL dispatch
579         ELSIF i = 1675 THEN
580           j := 1;
581           CALL dispatch
582         ELSIF i = 2092 THEN
583           j := 2;
584           CALL dispatch
585         ELSIF i = 2301 THEN
586           j := 3;
587           CALL dispatch
588         ELSIF i = 2703 THEN
589           j := 4;
590           CALL dispatch
591         ELSIF i = 3001 THEN
592           j := 5;
593           CALL dispatch
594         ELSIF i = 3289 THEN
595           j := 6;
596           CALL dispatch
597         ELSIF i = 3479 THEN
598           j := 7;
599           CALL dispatch
```

```
600         FI;
601         SKIP;
602         CALL T;
603         IF l = 1352 THEN
604           n := 0;
605           CALL dispatch
606         ELSIF l = 1675 THEN
607           n := 1;
608           CALL dispatch
609         ELSIF l = 2092 THEN
610           n := 2;
611           CALL dispatch
612         ELSIF l = 2301 THEN
613           n := 3;
614           CALL dispatch
615         ELSIF l = 2703 THEN
616           n := 4;
617           CALL dispatch
618         ELSIF l = 3001 THEN
619           n := 5;
620           CALL dispatch
621         ELSIF l = 3289 THEN
622           n := 6;
623           CALL dispatch
624         ELSIF l = 3479 THEN
625           n := 7;
626           CALL dispatch
627         FI
628       END
629       T ==
630         DO
631           SKIP;
632           DO
633             D_IF EVEN?(i) AND i <> 0 –>
```

```
634             i := i ** 2
635           [] ODD?(i) ->
636             i := i - 1
637           [] i = 0 ->
638             EXIT(1)
639           FI
640       OD
641     OD;
642     CALL U
643   END
644   U ==
645     IF i = 5 THEN
646       j := j + 2
647     ELSIF j = 7 THEN
648       j := j + 2
649     FI;
650     IF i = 5 THEN
651       k := k + 2
652     ELSIF j = 7 THEN
653       k := k + 2;
654       SKIP
655     FI;
656     CALL V
657   END
658   V ==
659     i := i + 5;
660     CALL W;
661     destination := 8232;
662     IF m = 231 THEN
663       n := 0;
664       CALL dispatch
665     ELSIF m = 312 THEN
666       n := 1;
667       CALL dispatch
```

```
668         ELSIF m = 478 THEN
669           n := 2;
670           CALL dispatch
671         ELSIF m = 567 THEN
672           n := 3;
673           CALL dispatch
674         ELSIF m = 624 THEN
675           n := 4;
676           CALL dispatch
677         ELSIF m = 765 THEN
678           n := 5;
679           CALL dispatch
680         ELSIF m = 879 THEN
681           n := 6;
682           CALL dispatch
683         ELSIF m = 904 THEN
684           n := 7;
685           CALL dispatch
686         FI
687       END
688       W ==
689         IF l > 500 AND l < 1000 THEN
690           IF m > 500 AND m < 1000 THEN
691             IF n > 500 AND m < 1000 THEN
692               destination := 3560;
693               i := -l;
694               j := -m;
695               k := -n
696             FI
697           FI
698         FI;
699         CALL X
700       END
701       X ==
```

```
702       IF i > n THEN
703         DO
704           WHILE i > n DO
705             k := k + n;
706             n := n - 1
707           OD;
708           EXIT(1)
709         OD
710       FI
711     END
712     Y ==
713       i := i + 1;
714       i := i + 1;
715       n := n + 1;
716       IF i + n ** 2 = 532 THEN
717         destination := 8651
718       ELSIF i + n ** 2 = 688 THEN
719         destination := 9899
720       ELSIF FALSE THEN
721         destination := 11438
722       ELSIF TRUE THEN
723         destination := 12219
724       ELSE
725         destination := 12623
726       FI;
727       CALL dispatch
728     END
729     dispatch ==
730       IF destination = 2526 THEN
731         CALL PROG
732       ELSIF destination = 3644 THEN
733         CALL B
734       ELSIF destination = 4677 THEN
735         CALL C
```

```
736         ELSIF destination = 6973 THEN
737           CALL D
738         ELSIF destination = 7099 THEN
739           CALL F
740         ELSIF destination = 7911 THEN
741           CALL G
742         ELSIF destination = 8232 THEN
743           CALL H
744         ELSIF destination = 8651 THEN
745           CALL J
746         ELSIF destination = 9016 THEN
747           CALL K
748         ELSIF destination = 9899 THEN
749           CALL L
750         ELSIF destination = 10211 THEN
751           i := 0;
752           CALL N
753         ELSIF destination = 10698 THEN
754           CALL O
755         ELSIF destination = 10945 THEN
756           CALL P
757         ELSIF destination = 11567 THEN
758           CALL S
759         ELSIF destination = 11601 THEN
760           CALL T
761         ELSIF destination = 12180 THEN
762           CALL U
763         ELSIF destination = 12219 THEN
764           CALL V
765         ELSIF destination = 12623 THEN
766           CALL W
767         ELSIF destination = 13381 THEN
768           CALL X
769         ELSE
```

```
770           CALL Z
771         FI
772       END
773     ENDACTIONS
774 ENDVAR
```

Listing C.2: Case Study 2: Final Program P_n WSL Code.

```
1  VAR <destination := 0,
2       i := 0,
3       j := 0,
4       k := 0,
5       l := 0,
6       m := 0,
7       n := 0>:
8    DO
9      FOR i := 0 TO 99 STEP 1 DO
10       j := j - 2
11     OD;
12     BEGIN
13       P1( VAR )
14     WHERE
15       PROC P1( VAR ) ==
16         IF k = 0 THEN
17           i := 1;
18           j := 1
19         FI;
20         P2( VAR )
21       END
22       PROC P2( VAR ) ==
23         k := 0;
24         WHILE i < 5 DO
25           i := i + 1;
26           j := 2 * i + j
27         OD;
```

```
28          P3( VAR )
29        END
30        PROC P3( VAR ) ==
31          m := 77;
32          n := i + j + 2;
33          IF (j < k OR i > j) AND l >= 25 THEN
34            ABORT;
35            EXIT(2)
36          ELSIF (i = j OR j = k) AND l >= 25 THEN
37            i := 33;
38            n := 2
39          ELSIF l >= 25 THEN
40            ACTIONS PROG2:
41              PROG2 ==
42                i := 0;
43                CALL AA
44              END
45              AA ==
46                IF l <> n THEN
47                  i := 1;
48                  CALL BB
49                FI;
50                i := -1;
51                CALL BB
52              END
53              BB ==
54                k := 11;
55                m := 23;
56                CALL CC
57              END
58              CC ==
59                l := 21;
60                CALL Z
61              END
```

```
62            ENDACTIONS
63          FI;
64          IF i <= j THEN
65            BEGIN
66              P1( VAR )
67            WHERE
68              PROC P1( VAR ) ==
69                m := 77;
70                n := i + j + 2;
71                IF l < 25 THEN
72                  m := 0;
73                  EXIT(2)
74                FI;
75                P2( VAR )
76              END
77              PROC P2( VAR ) ==
78                IF k = 0 THEN
79                  i := 1;
80                  j := 1
81                FI;
82                P3( VAR )
83              END
84              PROC P3( VAR ) ==
85                k := 0;
86                WHILE i < 5 DO
87                  i := i + 1;
88                  j := 2 * i + j
89                OD;
90                IF i <= 273 THEN
91                  IF l < 137 THEN
92                    k := l;
93                    l := l + 1;
94                    IF l < 137 THEN
95                      k := k + l;
```

```
96                      l := l + 1;
97                      IF l < 137 THEN
98                        k := k + l;
99                        l := l + 1;
100                       IF l < 137 THEN
101                         k := k + l;
102                         l := l + 1;
103                         WHILE l < 137 DO
104                           k := k + l;
105                           l := l + 1
106                         OD
107                       FI
108                     FI
109                   FI
110                 FI;
111                 m := 0;
112                 EXIT(2)
113               FI;
114               j := 0;
115               EXIT(2)
116             END
117           END
118         FI;
119         IF i > k THEN
120           IF i > l THEN
121             IF i > 77 THEN
122               IF i > n THEN
123                 EXIT(2)
124               FI;
125               DO
126                 DO
127                   IF i > n THEN
128                     EXIT(2)
129                   FI;
```

```
130                     i := i + 1;
131                     l := l - 1
132                   OD
133                 OD
134               FI
135             ELSE
136               DO
137                 j := i - 3;
138                 EXIT(1)
139               OD
140             FI
141           ELSE
142             DO
143               WHILE i > j DO
144                 j := j + 1;
145                 k := k - 1
146               OD;
147               EXIT(1)
148             OD
149           FI;
150           i := i + 1;
151           n := 72;
152           IF i > 72 THEN
153             ABORT
154           FI;
155           EXIT(2)
156       END
157     END
158   OD
159 ENDVAR
```

Appendix D

Case Study 3 WSL Code

Listing D.1: Case Study 3: Initial Program P_0 WSL Code.

```
1  BEGIN
2    A := ARRAY(10000, 0);
3    length := 0;
4    element := 0;
5    length := 10000;
6    element := 5001;
7    init := 1;
8    FOR i := 1 TO length STEP 1 DO
9      IF init = 1 THEN
10       IF FALSE THEN
11         length := 0;
12         element := 0
13       FI;
14       A[i] := length;
15       A[i] := A[i] * 2;
16       A[i] := A[i] - i;
17       A[i] := A[i] - i;
18       A[i] := A[i] + 2;
19     ELSIF init = 0 THEN
20       A[i] := length DIV 2;
21       A[i] := A[i] - 1
```

```
22      FI
23    OD;
24    SORT( VAR );
25    INCREASEALL( VAR );
26    SEARCH( VAR );
27    PRINT("RESULT: ", result)
28  WHERE
29    PROC SORT( VAR ) ==
30      n := 0;
31      swapped := 0;
32      n := length;
33      DO
34        swapped := 0;
35        FOR i := 1 TO n - 1 STEP 1 DO
36          n := length;
37          IF A[i] > A[i + 1] THEN
38            SWAP( VAR );
39            swapped := 1
40          FI
41        OD;
42        IF swapped = 0 THEN
43          EXIT(1)
44        FI
45      OD
46    END
47    PROC INCREASEALL( VAR ) ==
48      FOR i := 1 TO length STEP 1 DO
49        INCREASE( VAR )
50      OD
51    END
52    PROC SEARCH( VAR ) ==
53      low := 0;
54      high := 0;
55      result := 0;
```

```
56      low := 1;
57      high := length;
58      result := -1;
59      WHILE low <= high AND result = -1 DO
60        mid := high DIV 2;
61        mid := mid - low DIV 2;
62        mid := mid + low;
63        IF A[mid] > element THEN
64          high := mid - 1
65        ELSIF A[mid] < element THEN
66          low := mid + 1
67        ELSIF 5 > 7 THEN
68          result := mid
69        ELSIF FALSE THEN
70          ABORT
71        ELSE
72          result := mid
73        FI
74      OD
75    END
76    PROC SWAP( VAR ) ==
77      temp := 0;
78      temp := A[i];
79      A[i] := A[i + 1];
80      A[i + 1] := temp
81    END
82    PROC INCREASE( VAR ) ==
83      temp := 0;
84      temp := A[i] + 1;
85      A[i] := temp
86    END
87  END
```

Listing D.2: Case Study 3: Final Program P_n WSL Code.

```
1  A := ARRAY(10000, 0);
2  length := 10000;
3  element := 5001;
4  init := 1;
5  FOR i := 1 TO 10000 STEP 1 DO
6    A[i] := 10000;
7    A[i] := 2 * A[i];
8    A[i] := A[i] - i;
9    A[i] := A[i] - i;
10   A[i] := A[i] + 2
11 OD;
12 n := length;
13 DO
14   swapped := 0;
15   FOR i := 1 TO n - 1 STEP 1 DO
16     IF A[i + 1] < A[i] THEN
17       temp := A[i];
18       A[i] := A[i + 1];
19       A[i + 1] := temp;
20       swapped := 1
21     FI
22   OD;
23   IF swapped = 0 THEN
24     EXIT(1)
25   FI
26 OD;
27 FOR i := 1 TO length STEP 1 DO
28   temp := A[i] + 1;
29   A[i] := temp
30 OD;
31 low := 1;
32 high := length;
33 result := -1;
```

```
34 WHILE result = -1 AND high >= low DO
35   mid := -low DIV 2 + high DIV 2 + low;
36   IF A[mid] > element THEN
37     high := mid - 1
38   ELSIF A[mid] < element THEN
39     low := mid + 1
40   ELSE
41     result := mid
42   FI
43 OD;
44 PRINT("RESULT: ", result)
```